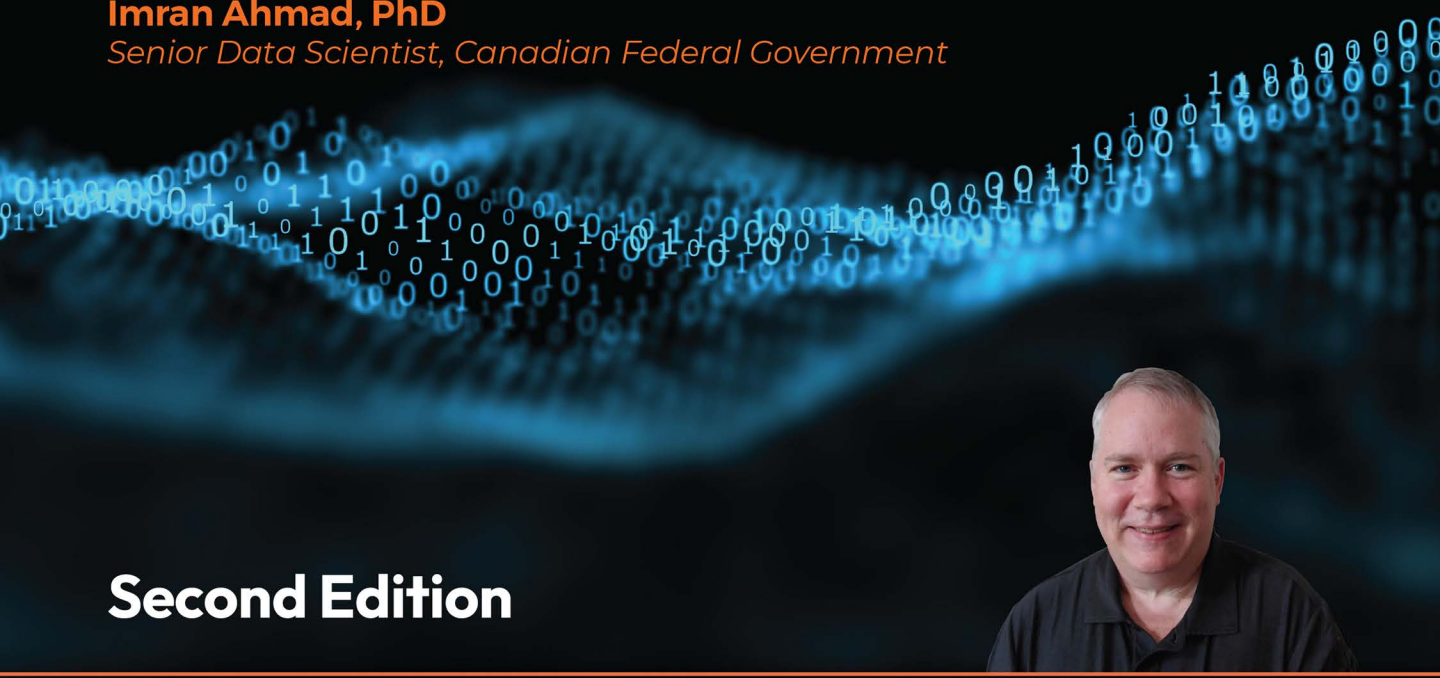
EXPERT INSIGHT

Build Your Own Programming Language

A programmer's guide to designing compilers, interpreters, and DSLs for modern computing problems

Foreword by:
Imran Ahmad, PhD
Senior Data Scientist, Canadian Federal Government



Second Edition

Clinton L. Jeffery

‹packt›

Build Your Own Programming Language

Second Edition

A programmer's guide to designing compilers, interpreters, and DSLs for modern computing problems

Clinton L. Jeffery

packt

BIRMINGHAM—MUMBAI

Build Your Own Programming Language

Second Edition

Senior Publishing Product Manager: Denim Pinto

Acquisition Editor: **Peer Reviews**: Gaurav Gavas

Project Editor: Parvathy Nair

Content Development Editor: Elliot Dallow

Copy Editor: Safis Editing

Technical Editor: Aneri Patel

Proofreader: Safis Editing

Indexer: Hemangini Bari

Presentation Designer: Ajay Patule

Developer Relations Marketing Executive: Vipanshu Pareshar

First published: December 2021

Second edition: January 2024

Production reference: 3150524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80461-802-8

www.packt.com

Foreword

In the dynamic world of computer science, the creation of a programming language stands as a testament to ingenuity and a deep understanding of computational principles. *Build Your Own Programming Language* is not just a guide; it is an invitation to delve into the complexity and beauty of programming language creation.

At the helm of this voyage is Clinton L. Jeffery, a distinguished professor and Chair of the Department of Computer Science and Engineering at the New Mexico Institute of Mining and Technology. His academic journey, marked by degrees from the University of Washington and the University of Arizona, has been a path of relentless exploration in the realms of programming languages, program monitoring, and visualization, among others. His work culminates in the creation of the Unicon programming language, a testament to his expertise and vision.

This book is structured to guide the reader through the nuanced process of developing a programming language. Beginning with motivations and types of language implementations, Jeffery sets the stage for understanding the fundamental "*why*" behind language design. He intricately discusses organizing a bytecode language and differentiates between programming languages and libraries, laying a solid foundation for both novices and experienced programmers.

The detailed chapters delve into the heart of language design, parsing, and the construction of syntax trees, with practical examples and case studies like the development of Unicon and the Jzero language. Jeffery's approach is meticulous, ensuring that readers grasp the essentials of technical requirements, lexical categories, context-free grammar, and symbol tables. This comprehensive coverage ensures that readers are not just following instructions but are truly understanding the principles at play.

What makes this book exceptional is its blend of theoretical knowledge and practical application. Jeffery does not shy away from the complexities of designing graphics facilities or tackling syntax trees and symbol tables. Instead, he embraces these challenges, guiding the reader with clarity and insight. The inclusion of questions at the end of each chapter prompts critical thinking and reflection, reinforcing the overall learning experience.

As you progress through *Build Your Own Programming Language*, you will find yourself not just acquiring knowledge, but also developing a new perspective on programming languages. They are not merely tools for tasks but are expressive mediums that reflect human creativity and problem-solving skills.

Clinton L. Jeffery, with his extensive experience and pioneering work in Unicon, provides a comprehensive and enlightening guide for anyone interested in the art and science of programming language development. Whether you are a student, a professional programmer, or an enthusiast of computer science, this book is a beacon, illuminating the path to understanding and creating your own programming language.

Welcome to a journey of discovery, creativity, and technical mastery in the world of programming languages!

Imran Ahmad, PhD

Senior Data Scientist, Canadian Federal Government

Contributors

About the author

Clinton L. Jeffery is Professor and Chair of the Department of Computer Science and Engineering at the New Mexico Institute of Mining and Technology. He received his B.S. from the University of Washington, and M.S. and Ph.D. degrees from the University of Arizona, all in computer science. He has conducted research and written many books and papers on programming languages, program monitoring, debugging, graphics, virtual environments, and visualization. With colleagues, he invented the Unicon programming language, hosted at `unicon.org`.

Steve Wampler, Sana Algaraibeh, and Phillip Thomas provided valuable feedback and suggestions for improving this book.

About the reviewers

Steve Wampler was awarded a Ph.D. in Computer Science by the University of Arizona. He has worked as an Associate Professor of Computer Science as well as a software designer for several major telescope projects, including the Gemini 8m telescopes project and the Daniel K Inouye solar telescope. He has been a software reviewer for a number of other major telescope systems and a technical reviewer for several other programming books.

Sana Algaraibeh was awarded her Ph.D. in Computer Science from the University of Idaho. She joined the faculty at the New Mexico Institute of Mining and Technology as a Computer Science instructor in 2022. Prior to that, she worked in academia for 14+ years as a lecturer, trainer, team leader, instructional designer, and Computer Science department chair at universities in Jordan and Saudi Arabia. She teaches Internet and Web Programming, Object-Oriented Programming, Python for Data Science, Algorithms and Data Structures, and Introduction to Programming. Her area of scholarship is computer science education and compiler error messages. She is interested in developing computational solutions integrated with modern pedagogy.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

`https://discord.com/invite/zGVbWaxqbw`

Table of Contents

Preface

This second edition was begun primarily at the suggestion of a first edition reader, who called me one day and explained that they were using the book for a programming language project. The project was not generating code for a bytecode interpreter or a native instruction set as covered in the first edition. Instead, they were creating a transpiler from a classic legacy programming language to a modern mainstream language. There are many such projects, because there is a lot of old code out there that is still heavily used. The Unicon translator itself started as a preprocessor and then was extended until it became in some sense, a transpiler. So, when Packt asked for a second edition, it was natural to propose a new chapter on that topic; this edition has a new *Chapter 11* and all chapters (starting from what was *Chapter 11* in the previous edition) have seen their number incremented by one. A second major facet of this second edition was requested by Packt and not my idea at all. They requested that the IDE syntax coloring chapter be extended to deal with the topic of adding syntax coloring to mainstream IDEs that I did not write and do not use, instead of its previous content on syntax coloring in the Unicon IDEs. Although this topic is outside my comfort zone, it is a valuable topic that is somewhat under-documented at present and easily deserves inclusion, so here it is. You, as the reader, can decide whether I have managed to do it any justice as an introduction to that topic.

After 60+ years of high-level language development, programming is still too difficult. The demand for software of ever-increasing size and complexity has exploded due to hardware advances, while programming languages have improved far more slowly. Creating new languages for specific purposes is one antidote for this software crisis.

This book is about building new programming languages. The topic of programming language design is introduced, although the primary emphasis is on programming language implementation. Within this heavily studied subject, the novel aspect of this book is its fusing of traditional compiler-compiler tools (Flex and Byacc) with two higher-level implementation languages. A very high-level language (Unicon) plows through a compiler's data structures and algorithms like butter, while a mainstream modern language (Java) shows how to implement the same code in a more typical production environment.

One thing I didn't really understand after my college compiler class was that the compiler is only one part of a programming language implementation. Higher-level languages, including most newer languages, may have a runtime system that dwarfs their compiler. For this reason, the second half of this book spends quality time on a variety of aspects of language runtime systems, ranging from bytecode interpreters to garbage collection.

Who this book is for

This book is for software developers interested in the idea of inventing their own language or developing a domain-specific language. Computer science students taking compiler construction courses will also find this book highly useful as a practical guide to language implementation to supplement more theoretical textbooks. Intermediate-level knowledge and experience of working with a high-level language such as Java or C++ are required in order to get the most out of this book.

What this book covers

Chapter 1, *Why Build Another Programming Language?*, discusses when to build a programming language, and when to instead design a function library or a class library. Many readers of this book will already know that they want to build their own programming language. Some should design a library instead.

Chapter 2, *Programming Language Design*, covers how to precisely define a programming language, which is important to know before trying to build a programming language. This includes the design of the lexical and syntax features of the language, as well as its semantics. Good language designs usually use as much familiar syntax as possible.

Chapter 3, *Scanning Source Code*, presents lexical analysis, including regular expression notation and the tools Ulex and JFlex. By the end, you will be opening source code files, reading them character by character, and reporting their contents as a stream of tokens consisting of the individual words, operators, and punctuation in the source file.

Chapter 4, *Parsing*, presents syntax analysis, including context-free grammars and the tools `iyacc` and `byacc/j`. You will learn how to debug problems in grammars that prevent parsing, and report syntax errors when they occur.

Chapter 5, *Syntax Trees*, covers syntax trees. The main by-product of the parsing process is the construction of a tree data structure that represents the source code's logical structure. The construction of tree nodes takes place in the semantic actions that execute on each grammar rule.

Chapter 6, *Symbol Tables*, shows you how to construct symbol tables, insert symbols into them, and use symbol tables to identify two kinds of semantic errors: undeclared and illegally redeclared variables. In order to understand variable references in executable code, each variable's scope and lifetime must be tracked. This is accomplished by means of table data structures that are auxiliary to the syntax tree.

Chapter 7, *Checking Base Types*, covers type checking, which is a major task required in most programming languages. Type checking can be performed at compile time or at runtime. This chapter covers the common case of static compile-time type checking for base types, also referred to as atomic or scalar types.

Chapter 8, *Checking Types on Arrays, Method Calls, and Structure Accesses*, shows you how to perform type checks for the arrays, parameters, and return types of method calls in the Jzero subset of Java. The more difficult parts of type checking are when multiple or composite types are involved. This is the case when functions with multiple parameter types must be checked, or when arrays, hash tables, class instances, or other composite types must be checked.

Chapter 9, *Intermediate Code Generation*, shows you how to generate intermediate code by looking at examples for the Jzero language. Before generating code for execution, most compilers turn the syntax tree into a list of machine-independent intermediate code instructions. Key aspects of control flow, such as the generation of labels and goto instructions, are handled at this point.

Chapter 10, *Syntax Coloring in an IDE*, addresses the challenge of incorporating information from syntax analysis into an IDE in order to provide syntax coloring and visual feedback about syntax errors. A programming language requires more than just a compiler or interpreter – it requires an ecosystem of tools for developers. This ecosystem can include debuggers, online help, or an integrated development environment.

Chapter 11, *Preprocessors and Transpilers*, gives an overview of generating output intended to be compiled or interpreted by another high-level language. Preprocessors are usually line-oriented and translate lines into very similar output, while transpilers usually translate one high-level language to a different high-level language with a full parse and significant semantic changes.

Chapter 12, *Bytecode Interpreters*, covers designing the instruction set and the interpreter that executes bytecode. A new domain-specific language may include high-level domain programming features that are not supported directly by mainstream CPUs. The most practical way to generate code for many languages is to generate bytecode for an abstract machine whose instruction set directly supports the domain, and then execute programs by interpreting that instruction set.

Chapter 13, *Generating Bytecode*, continues with code generation, taking the intermediate code from *Chapter 9*, *Intermediate Code Generation*, and generating bytecode from it. Translation from intermediate code to bytecode is a matter of walking through a giant linked list, translating each intermediate code instruction into one or more bytecode instructions. Typically, this is a loop to traverse the linked list, with a different chunk of code for each intermediate code instruction.

Chapter 14, *Native Code Generation*, provides an overview of generating native code for x86_64. Some programming languages require native code to achieve their performance requirements. Native code generation is like bytecode generation, but more complex, involving register allocation and memory addressing modes.

Chapter 15, *Implementing Operators and Built-In Functions*, describes how to support very high-level and domain-specific language features by adding operators and functions that are built into the language. Very high-level and domain-specific language features are often best represented by operators and functions that are built into the language, rather than library functions. Adding built-ins may simplify your language, improve its performance, or enable side effects in your language semantics that would otherwise be difficult or impossible. The examples in this chapter are drawn from Unicon, as it is much higher level than Java and implements more complex semantics in its built-ins.

Chapter 16, *Domain Control Structures*, covers when you need a new control structure, and provides example control structures that process text using string scanning, and render graphics regions. The generic code in previous chapters covered basic conditional and looping control structures, but domain-specific languages often have unique or customized semantics for which they introduce novel control structures. Adding new control structures is substantially more difficult than adding a new function or operator, but it is what makes domain-specific languages worth developing instead of just writing class libraries.

Chapter 17, *Garbage Collection*, presents a couple of methods with which you can implement garbage collection in your language. Memory management is one of the most important aspects of modern programming languages, and all the cool programming languages feature automatic memory management via garbage collection. This chapter provides a couple of options as to how you might implement garbage collection in your language, including reference counting, and mark-and-sweep garbage collection.

Chapter 18, *Final Thoughts*, reflects on the main topics presented in the book and gives you some food for thought. It considers what was learned from writing this book and gives you many suggestions for further reading.

Appendix, *Unicon Essentials*, describes enough of the Unicon programming language to understand those examples in this book that are in Unicon. Most examples are given side by side in Unicon and Java, but the Unicon versions are usually shorter and easier to read.

Answers, gives you some proposed answers to the revision questions placed at the end of each chapter.

To get the most out of this book

In order to understand this book, you should be an intermediate-or-better programmer in Java or a similar language; a C programmer who knows an object-oriented language will be fine.

Software/hardware covered in the book	Operating system requirements
Unicon 13.2, Uflex, and Iyacc	Windows, Linux
Java, Jflex, and Byacc/J	Windows, Linux
GNU Make	Windows, Linux

Instructions for installing and using the tools are spread out a bit to reduce the startup effort, appearing in *Chapter 3*, *Scanning Source Code*, to *Chapter 5*, *Syntax Trees*. If you are technically gifted, you may be able to get all these tools to run on macOS, but it was not used or tested during the writing of this book.

NOTE

If you are using the digital version of this book, we advise you to type the code yourself or, better yet, access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Build-Your-Own-Programming-Language-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

The Code in Action videos for this book can be viewed at `https://bit.ly/3njc15D`.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781804618028`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and X (more commonly known as Twitter) handles. For example: "The `JSRC` macro gives the names of all the Java files to be compiled."

A block of code is set as follows:

```
public class address {
   public String region;
   public int offset;
   address(String s, int x) { region = s; offset = x; }
}
```

Any command-line input or output is written as follows:

```
j0 hello.java              java ch9.j0 hello.java
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "A `makefile` is like a `lex` or `yacc` specification, except instead of recognizing patterns of strings, a `makefile` specifies a graph of **build dependencies** between files".

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Build Your Own Programming Language, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review` page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804618028`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Section I

Programming Language Frontends

In this section, you will create a basic language design and implement the frontend of a compiler for it, including a lexical analyzer and a parser that builds a syntax tree from an input source file.

This section comprises the following chapters:

- *Chapter 1*, *Why Build Another Programming Language?*
- *Chapter 2*, *Programming Language Design*
- *Chapter 3*, *Scanning Source Code*
- *Chapter 4*, *Parsing*
- *Chapter 5*, *Syntax Trees*

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 that kind of effort. Do you have the patience and persistence that it takes?

This chapter points out a few good reasons to build your own programming language, as well as some circumstances in which you don't *need* to build your contemplated language. After all, designing a class library for your application domain is often simpler and just as effective. However, libraries have their limitations, and sometimes, only a new language will do.

After this chapter, the rest of this book will take for granted that, having considered things carefully, you have decided to build a language. But first, we're going to consider our initial options by covering the following main topics in this chapter:

- Motivations for writing your own programming language
- Types of programming language implementations
- Organizing a bytecode language implementation
- Languages used in the examples
- The difference between programming languages and libraries
- Applicability to other software engineering tasks

- Establishing the requirements for your language
- Case study – requirements that inspired the Unicon language

Let's start by looking at motivations.

Motivations for writing your own programming language

Sure, some programming language inventors are rock stars of computer science, such as Dennis Ritchie or Guido van Rossum! Becoming a rock star in computer science was easier back in the previous century. In 1993, I heard the following report from an attendee of the second ACM 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 two later when Java hit the scene, and perhaps a dozen times since then when important languages such as Go emerged. After a mere six decades, it would be unwise to claim our field is mature and that there's nothing new to invent that might make you famous.

In any case, 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 a desire to know how things work are valid reasons, so long as you've got the time and inclination, but perhaps the best reason to build your own programming language is 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. This is the stuff Master's theses and PhD dissertations are made of. Every once in a blue moon, someone comes up with a whole new style of computing for which a new programming paradigm requires a new language.

While we are discussing your motivations for building a language, let's also talk about the different kinds of languages, how they are organized, and the examples this book will use to guide you.

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, which is to say, the 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 the 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 in Java

The first option is fun, but the resulting language is usually too slow to satisfy real-world project requirements. The second option is often optimal, but may be too labor-intensive; a good native compiler may take years of effort.

The third option is by far the easiest and probably the most fun, and I have used it before with good success. Don't discount a transpiler implementation as a kind of cheating, but do be aware that it has its problems. The first version of C++, AT&T's cfront tool, was a transpiler, but that gave way to compilers, and not just because cfront 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 cuts.

For the most part, this book focuses on the fourth option; over the course of several chapters, we will build a bytecode compiler with an accompanying bytecode machine because that is a sweet spot that gives a lot of flexibility, while still offering decent performance. A chapter on transpilers and preprocessors is provided for those of you who may prefer to implement your language by generating code for another high-level language. A chapter on native code compilation is also 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 implement 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 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.

In addition to the steps of this bytecode virtual machine compiler, a **bytecode interpreter** is written to load and execute programs. It is a giant loop with a switch statement in it. For very high-level programming languages, the compiler might be no big deal, and all the magic may be in the bytecode interpreter. The whole organization can be summarized by the following diagram:

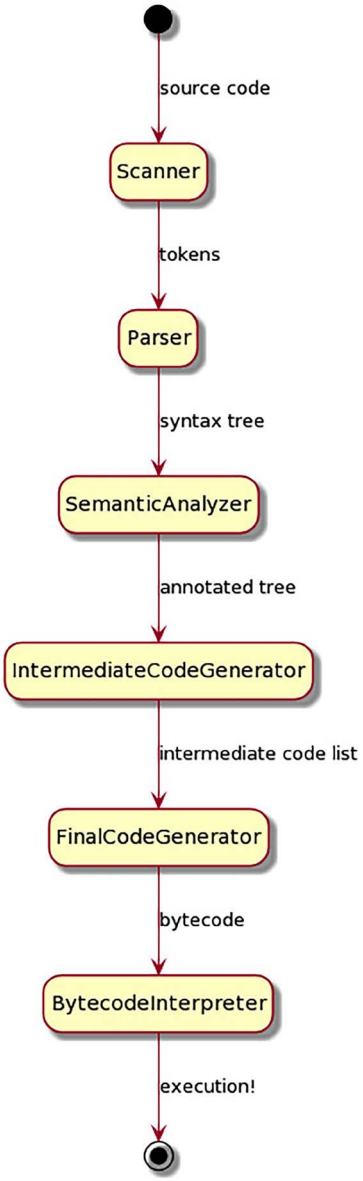


Figure 1.1: Phases and dataflow in a simple programming language

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, as well as 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 Java (or C++, 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 programming languages. As many examples as possible are provided in both languages, and the examples in the two languages are written as similarly as possible. Sometimes, this will be to the advantage of Java, which is a bit lower level than Unicon. There are sometimes fancier or shorter ways to write things in Unicon, but our Unicon examples will stick as close to Java as possible. The differences between Java and Unicon will be obvious, but they are somewhat lessened in importance by the compiler construction tools we will use.

This book uses modern descendants of the venerable Lex and YACC tools to generate our scanner and parser. Lex and YACC are declarative programming languages that solve some of our hard problems at a higher level than Java or Unicon. It would have been nice if a modern descendant of Lex and YACC (such as ANTLR) supported both Java and Unicon, but such is not the case. One of the very cool parts of this book is this: by choosing tools for Java and Unicon that are very compatible with the original Lex and YACC and extending them a bit, we have managed to use the same lexical and syntax specifications of our compiler in both Java and Unicon!

While Java and Unicon are our implementation languages, we 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, this book introduces 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 is 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 whether someone had already posted a Jzero definition 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 Java 21; maybe other recent versions of Java will work. You can get OpenJDK from `http://openjdk.org`, 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 13.3, 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 should follow the instructions found on the `unicon.org` site.

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

The difference between programming languages and libraries

Unless you are in it for the "fun" or the intellectual experience, building a programming language is a lot of work that might not be necessary. If your motives are strictly utilitarian, you don't have to 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) to manipulate 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, so 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.
- Applications that use libraries can become broken if the library changes incompatibly in a later version.

There is a natural evolutionary path from a library to a 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 tools and techniques to implement programming languages is useful in many 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, interpreters, and the 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 up front, 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 into Python gets compiled into bytecode and executed by a bytecode machine, rather than being interpreted directly.

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) your language 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.

In addition to identifying functional and non-functional requirements, many requirements analysis approaches 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 and perform such a use case analysis. 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 presents the implementation of a language called Jzero, here are some requirements for Jzero. Some of these requirements may appear arbitrary. You could certainly add your own requirements and produce your own Java dialect, but this list describes what we are aiming for in this book. If it is not clear to you where one of the following requirements 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 provide enough features 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. However, it only needs to support a subset of their functionality, (as you'll see later). These types are enough 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 that execute source code directly without any internal code generation step are a very retro thing, evocative of the 1960s and 1970s. They tend to execute unacceptably slowly by modern standards. On the other hand, you might very well decide that your language should provide the highly interactive look and feel of a pure interpreter, like Python does. Anyhow, that is not in Jzero's requirements.

- Jzero should be as simple as possible so that I can explain it. Sadly, this rules out writing a full description of a native code generator or even an implementation that targets 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, rather 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 facilities and is used to write 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 its integration of goal-directed evaluation, including backtracking and automatic resumption of generators, into a familiar mainstream syntax. This leads us to Unicon's first requirement.

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**. A generator is an expression that is capable of computing more than one result; several popular languages feature generators. Goal-directed evaluation is a semantic to execute code in which expressions either succeed or fail, and when they fail, generators within the expression can be resumed to try alternative results that might make the whole expression succeed. This is a big topic beyond the scope of this section, but if you want to learn more, you can check out *The Icon Programming Language, Third Edition*, by Ralph and Madge Griswold, at `www.cs.arizona.edu/icon`.

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 preservation goal is 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 as part of requirement #1. We have placed a much firmer 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. Unicon requirement #2 was to support programming in large-scale projects.

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

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 modern programmers are often required to write much larger programs than Icon was designed to handle.

For this reason of scalability, 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. 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

Icon was designed for classic UNIX pipe-and-filter text processing of local files. Over time, more and more people wanted to use it to write programs that required more sophisticated forms of input/output, such as networking or graphics.

Arguably, despite billionfold improvements in CPU speed and memory size, the biggest difference between programming in 1970 and programming in the 2020s 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.

Support for I/O is a moving target. At first, with Unicon, I/O consisted of 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. The definition of what I/O capabilities are ubiquitous continues to evolve, varying by platform, but touch input and gestures or shader programming capabilities are examples of things that have become ubiquitous today, and maybe they should be added to the Unicon language as part of this requirement. The challenge posed by this requirement is increased by Unicon requirement #4.

Unicon requirement #4 – provide universally implementable system interfaces

Icon is very portable. I have run it 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.

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, to meet this requirement, Unicon should be ported to support Android and iOS, if 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. However, 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 for your language than 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 for the programmers that use your language, as well as what hardware and software platforms it must run on. The look and feel of a programming language include 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, which 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.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

`https://discord.com/invite/zGVbWaxqbw`

2

Programming Language Design

Before trying to build a programming language, you need to define it. This includes the design of the features of the language that are visible on its surface, including basic rules to form words and punctuation. This also includes higher-level rules, called **syntax**, that govern the number and order of words and punctuation in larger chunks of programs, such as expressions, statements, functions, classes, packages, and programs. Language design also includes the underlying meaning, also known as **semantics**.

Programming language design often begins with you writing example code to illustrate each of the important features of your language, as well as show the variations that are possible for each construct. Writing examples with a critical eye lets you find and fix many possible inconsistencies in your initial ideas. From these examples, you can then capture the general rules that each language construct follows. Write down sentences that describe your rules as you understand them from your examples. Note that there are two kinds of rules. **Lexical rules** govern what characters must be treated together, such as words, or multi-character operators, such as ++. **Syntax rules**, on the other hand, are rules to combine multiple words or punctuation to form a larger meaning; in natural language, they are often phrases, sentences, or paragraphs, while in a programming language, they might be expressions, statements, functions, or programs.

Once you have come up with examples of everything that you want your language to do, and have written down the lexical and syntax rules, it is time to write a language design document (or language specification) to which you can refer while implementing your language. You can change things later, but it helps to have a plan to work from.

In this chapter, we're going to cover the following main topics:

- Determining the kinds of words and punctuation to provide in your language
- Specifying the control flow
- Deciding on what kinds of data to support
- An overall program structure
- Completing the Jzero language definition
- Case study – designing graphics facilities in Unicon

Let's start by identifying the basic elements that are allowed in the source code in your language.

Determining the kinds of words and punctuation to provide in your language

Programming languages have several different categories of words and punctuation. In natural language, words are categorized into **parts of speech** – nouns, verbs, adjectives, and so on. The categories that correspond to the parts of speech that you will have to invent for a programming language can be constructed by doing the following:

- Defining a set of **reserved words** or keywords
- Specifying characters in **identifiers** that name variables, functions, and constants
- Creating a format for **literal** constant values for built-in data types
- Defining single and multi-letter **operators** and punctuation marks

You should write down precise descriptions of each of these categories as part of your language design document. In some cases, you might just make lists of particular words or punctuation to use, but in other cases, you will need patterns or some other way to convey what is and is not allowed in a category.

For reserved words, a list will do for now. For names of things, a precise description must include details such as what non-letter symbols are allowed in such names. For example, in Java, names must begin with a letter and can then include letters and digits; underscores are allowed and treated as letters. In other languages, hyphens are allowed within names, so the three symbols `a`, `-`, and `b` make up a valid name, not a subtraction of `b` from `a`. When a precise description fails, a complete set of examples will suffice.

Constant values, also called **literals**, are a surprising and major source of complexity in lexical analyzers. Attempting to precisely describe real numbers in Java comes out something like this: Java has two different kinds of real numbers – floats and doubles – but they look the same until you get to the end, where there is an optional `f` (or `F`) or `d` (or `D`) to distinguish floats from doubles. Before that, real numbers must have either a decimal point (`.`), an exponent (`e` or `E`) part, or both. If there is a decimal point, there must be at least one digit on one side of the decimal or the other. If there is an exponent part, it must have an `e` (or `E`), followed by an optional minus sign and one or more digits. To make matters worse, Java has a weird hexadecimal real constant format, consisting of `0x` or `0X` followed by digits in hex format, with an optional mantissa consisting of a period followed by hexadecimal digits, and a mandatory power part consisting of a `p` (or `P`), followed by digits in the decimal format that multiplies the number by 2, raised to that power. If you want to write constants like 0x3.0fp8, then this IEEE-based format is for you.

Describing operators and punctuation marks is usually almost as easy as listing the reserved words. One major difference between operators and punctuation marks is that operators usually have **precedence** rules that you will need to determine. For example, in numeric processing, the multiplication operator has almost always higher precedence than the addition operator, so `x + y * z` will multiply `y * z` before it adds `x` to the product of `y` and `z`. In most languages, there are at least three to five levels of precedence, and many popular mainstream languages have from 13 to 20 levels of precedence that must be considered carefully.

The following diagram shows the operator precedence table for Java, from the lowest to highest precedence. We will need it for Jzero:

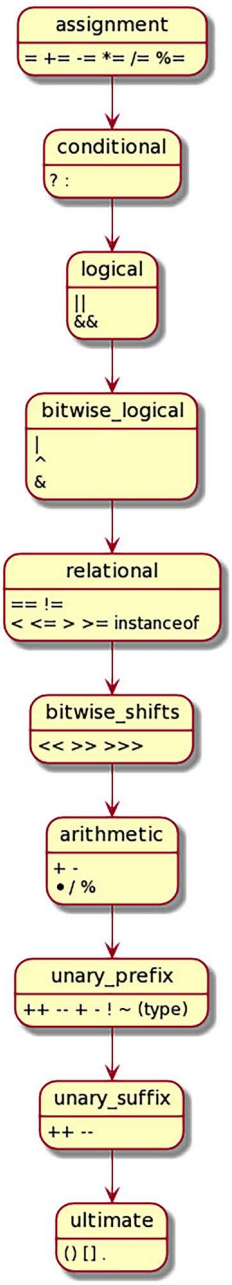


Figure 2.1: Java operator precedence

The preceding diagram shows that Java has a lot of operators, organized into 10 levels of precedence, although I might be simplifying this a bit. In your language, you might get away with fewer, but you will have to address the issue of operator precedence if you want to build a real language.

A similar issue is operator **associativity**. In many languages, most operators associate from left to right, but a few unusual ones associate from right to left. For example, the `x + y + z` expression is equivalent to `(x + y) + z`, but the `x = y = 0` expression is equivalent to `x = (y = 0)`.

The principle of least surprise applies to operator precedence and associativity, as well as to what operators you put in your language in the first place. If you define arithmetic operators and give them unusual precedence or associativity, people will reject your language out of hand. If you happen to be introducing new, possibly domain-specific data types, you have way more freedom to define operator precedence and associativity for any new operators you introduce in your language for those types.

Once you have determined what the individual words and punctuation in your language should be, you can work your way up to larger constructs. This is the transition from lexical analysis to syntax, and syntax is important because it is the level at which bits of code become large enough to specify some computation to be performed. We will look at this in more detail in later chapters, but at the design stage, you should at least think about how programmers will specify the control flow, declare data, and build entire programs. First, you must plan for the control flow.

Specifying the control flow

The **control flow** is how a program's execution proceeds from place to place within the source code. Most control flow constructs should be familiar to programmers who have been trained in mainstream programming languages. The innovations in your language design can then focus on the features that are novel or domain-specific, and that motivated you to create a new language in the first place. Make these novel things as simple and readable as possible. Envision how those new features ought to fit into the rest of the programming language.

Every language must have conditionals and loops, and almost all of them use `if` and `while` to start them. You could invent your own special syntax for an `if` expression, but unless you've got a good reason to, you would be shooting yourself in the foot. Here are some control flow constructs from Java that would certainly be in Jzero:

```
if (e) s;
if (e) s1 else s2;
while (e) s;
for (…) s;
```

Here are some other less common Java control flow constructs that are not in Jzero. If they were to appear in a program, what should a Jzero compiler do with them?

```
switch (e) { … }
do s while (e);
```

Since these constructs are not in Jzero, if they appear in the input source code, then by default, our compiler will print a cryptic syntax error message that doesn't explain things very well. In the next two chapters, we will make our compiler for Jzero print a nice error message about the Java features that it does not support.

Besides conditionals and loops, languages tend to have a syntax to call subroutines and return afterward. All these ubiquitous forms of control flow are abstractions of the underlying machine's capability to change the location where instructions are executing – the `GOTO`. If you invent a better notation for changing the location where instructions are executing, it will be a big deal.

The biggest controversy when designing many or most control flow constructs seems to be whether they are **statements**, or whether you should make them **expressions** that produce a result that can be used in a surrounding expression. I have used languages where the result of `if` expressions are useful – C/C++/Java even have an operator for that: the `i?t:e` conditional operator. I have not found a language that did something very meaningful by making a `while` loop an expression; the best the languages did was to have the `while` expressions produce a result, telling us whether the loop exited due to the test condition or an internal break.

If you are inventing a new language from scratch, one of the big questions for you is whether you should come up with some new control structure(s) to support your intended application domain. For example, suppose you want your language to provide special support for investing in the stock market. If you manage to come up with a better control structure for specifying conditions, constraints, or iterative operations within this domain, you might provide a competitive edge to those who are coding in your language for this domain. The program will have to run on an underlying von Neuman instruction set, so you will have to figure out how to map any such new control structure to instructions such as Boolean logic tests and `GOTO` instructions.

Whatever control flow constructs you decide to support, you will also need to design a set of data types and declarations that reflect the information that the programs in your language will manipulate.

Deciding on what kinds of data to support

There are at least three categories of data types to consider in your language design. We will describe each of these in this section. The first one is atomic, scalar primitive types, often called first-class data types. The second is composite or container types, which hold and organize collections of values. The third (which may be variants of the first or second categories) is application domain-specific types. You should formulate a plan for each of these categories.

Atomic types

Atomic types are generally built-in and immutable. As the word immutable suggests, you cannot modify existing atomic values, only combine them to compute new values. Pretty much all languages have such built-in atomic types for numbers and a few additional types. A Boolean type, `null` type, and maybe a `string` type are common atomics, but some languages have others.

You decide just how complicated to get with atomics: how many different machine representations of integers and real numbers do programs written in your language need? Some higher-level languages such as BASIC might provide a single type for all numbers, while lower-level languages such as C or C++ might provide 5 or 10 (or more) representations for different sizes and kinds of integers, and another few for real numbers. The more you add, the more flexibility and control you give to programmers who use your language, but the more difficult your implementation task will be later. In addition, the increased complexity reduces readability and makes programs harder to understand.

Similarly, it is impossible to design a single string data type that is ideal for all applications that use strings a lot. But how many string types do you want to support? One extreme is having no string type at all, only a short integer type to hold characters. Such languages consider strings to be composite types. Maybe strings are supported only by a library rather than in the language. Strings may be arrays or objects, but even such languages usually have some special lexical rules that allow string constant values to be given as double-quoted sequences of characters of some kind. Another extreme is that, given the importance of strings in many application domains, your language might want to support multiple string types for various character representations (ASCII, UTF8, and so on) with auxiliary types (character sets) and special types and control structures that support the analysis and construction of strings. Many popular languages treat strings as a special atomic type.

If you are especially clever, you may decide to support only a few built-in types for numbers and strings but make those types as flexible as possible. Once you go beyond integers, real numbers, and strings, the only types that are universal are container types, which allow you to assemble data structures.

Some of the things you must think about regarding atomic types include the following:

- How many values do they have?
- How are all those values encoded as literal constants in the source code?
- What kinds of operators or built-in functions use this type as operands or parameters?

The first question will tell you how many bytes the type will require in memory. The second and third questions tie back to determining the rules for words and punctuation in the language. The third question may also give insight into how much effort, in terms of the code generator or run-time system, will be required to implement support for the type in your language. Atomic types can be more work or less work to implement, but they are seldom as complicated as composite types, which we will discuss next.

Composite types

Composite types are types that help you allocate and access multiple values in a coordinated fashion. Languages vary enormously regarding the extent of their syntax support for composite types. Some only support arrays and structs (Java programmers: you can think of these as classes without methods) and require programmers to build all their own data structures on top of these. Many provide all higher-level composite types via libraries. However, some higher-level languages provide numerous sophisticated data structures as built-ins with syntax support.

The most ubiquitous composite type is an **array** type, where multiple values are accessed using a numerically contiguous range of integer indices. You will probably have something like an array in your language. Your main design considerations should be how the indices are given, and how changes in the size of the composite value are handled. Most popular languages use indices that start at zero. Zero-based array indexes simplify index calculations and are easier for a language inventor to implement, but they are less intuitive for new programmers. Some languages use 1-based indices or allow a programmer to specify a range of indices, starting at an arbitrary integer other than 0.

Regarding changes in size, some languages allow no changes in size at all in their array types, or they make the programmer jump through hoops to build new arrays of different sizes based on existing arrays.

Other languages are engineered to make adding values to an array a cheap and easy operation. No one design is perfect for all applications, so you just pick one and live with the consequences: do you choose to support multiple array-like data types for different purposes, or instead choose to design a very clever type that accommodates a range of common uses well?

Besides arrays, you should think about what other composite types you need. Almost all languages support a record, struct, or class type to group values of several different types together and access them by names, called fields. The more elaborate you get with this, the more complex your language implementation will be. If you need proper object orientation in your language, be prepared to pay for it in time spent writing your compiler and runtime code. Features like classes and inheritance do not come for free. Language designers are advised to keep things simple, but as a programmer, I would not want to use a programming language that did not give me this capability in some form.

You might be able to think of several other composite types that are essential for your language, which is great, especially if they will be used a lot in the programs that you care about. I will talk about one more composite type that is of great practical value: the (hash) **table** data type, also commonly called a **dictionary** type. A table type is something halfway in between an array and a record type. You index values using names, and these names are not fixed; new names can be computed while the program runs. Any modern language that omits this type is just leaving many of its prospective users out. For this reason, your language may want to include a table type. Composite types are general-purpose "glue" that's used to assemble complex data structures, but you should also consider whether some special-purpose types, either atomic or composite, belong in your language to support applications that are difficult to write in general-purpose languages.

Domain-specific types

In addition to whatever general-purpose atomic and composite types you decide to include, you should think about whether your programming language is aimed at a domain-specific niche; if so, what data types can your language include to support that domain? There is a smooth continuum between domain-specific languages that provide **domain-specific** types and control structures and general-purpose languages such as C++ and Java, which provide libraries for everything. Class libraries are powerful, but for some applications and domains, the library approach may be more complex and bug-prone than a language expressly designed to support the domain. For example, Java and C++ have string classes, but they do not support complex text-processing applications better than languages that have special-purpose types and control structures for string processing. Besides data types, your language design will need an idea of how programs are assembled and organized.

Overall program structure

When looking at the overall program structure, we need to look at how entire programs are organized and put together, as well as the important question of how much nesting is in your language. It almost seems like an afterthought, but how and where will the source code in programs begin executing? In languages based on C, execution starts from a `main()` function, while in scripting languages, the source code is executed as it is read in, so there is no need for a `main()` function to start the ball rolling.

Program structure also raises the basic question of whether a whole program must be translated and run together, or if different packages, classes, or functions can be separately compiled and then linked and/or loaded together for a program to run. A language inventor can dodge a lot of implementation complexity by either building things into the language (if it is built in, there is no need to figure out linking), requiring the whole program's source code to be presented at runtime, or by generating code for some well-known standard execution format where someone else's linker and loader will do all the hard work.

Perhaps the biggest design question relating to the overall program structure is which constructs may be nested, and what limits on nesting are present, if any. This is perhaps best illustrated by an example. Once upon a time, two obscure languages were invented around 1970 that struggled for dominance: **C** and **Pascal**.

The C language was almost flat – a program was a set of functions linked together, and only relatively small (fine-grained) things could be nested: expressions, statements, and, reluctantly, struct definitions.

In contrast, the Pascal language was fabulously more nested and recursive. Almost everything could be nested. Notably, functions could be embedded within functions, arbitrarily deep. Although C and Pascal were roughly equivalent in power, and Pascal had a bit of a head start and was by far the most popular in university courses, C eventually won. Why? There are many contributing factors that might explain why C won out over Pascal. One factor might be that nesting functions adds complexity without adding much value.

Because C won, many modern mainstream languages (I am thinking especially of C++ and Java here) started almost flat. But over time, they have added more and more nesting. Why is this? Perhaps it is natural for programming languages to add features over time until they are very complex. Niklaus Wirth saw this coming and advocated for a return to smallness and simplicity in software, but his pleas largely fell on deaf ears, and his languages support lots of nesting too.

What is the practical upshot for you, as a budding language designer? Don't over-engineer your language. Keep it as simple as possible. Don't nest things unless they need to be nested. And be prepared to pay (as a language implementor) every time you ignore this advice!

Now, it's time to draw a few programming language design examples from Jzero and Unicon. In the case of Jzero, since it is a subset of Java, the design is either a big nothingburger (we use Java's design) or it is subtractive: what do we take away from Java to make Jzero, and what will that look and feel like? Despite early efforts to keep it small, Java is a large language. If, as part of our design, we make a list of everything that is in Java that is not in Jzero, it will be a long list.

Due to the constraints of page space and programming time, Jzero must be a tiny subset of Java. However, ideally, any legal Java program that is input to Jzero would not fail embarrassingly – it would either compile and run correctly, or it would print a useful explanatory message that conveys what Java feature(s) are being used that Jzero does not support. So that you can easily understand the rest of this book, as well as to help keep your expectations to a manageable size, the next section will cover additional details regarding what is in Jzero and what is not.

Completing the Jzero language definition

In the previous chapter, we listed the requirements for the language that will be implemented in this book, and the previous section elaborated on some of its design considerations. For reference purposes, this section will describe additional details regarding the Jzero language. If you find any discrepancies between this section and our Jzero compiler, then they are bugs. Programming language designers use more precise formal tools to define various aspects of a language; notations to describe lexical and syntax rules will be presented in the next two chapters. This section will describe the language in layman's terms.

A Jzero program consists of a single class in a single file. This class may consist of multiple methods and variables, but all of them are **static**. A Jzero program starts by executing a static method called `main()`, which is required. The kinds of statements that are allowed in Jzero are assignment statements, `if` statements, `while` statements, and the invocation of `void` methods. The kinds of expressions that are allowed in a Jzero program include arithmetic, relational, and Boolean logic operators, as well as the invocation of non-void methods.

The Jzero language supports the `bool`, `char`, `int`, and `long` atomic types. The `int` and `long` types are equivalent to `64`-bit integer data types.