CHAPTER 1

Introduction

The objective of this book is to empower you to create new computer languages. This book pursues its goal even though hundreds of vital computer languages are alive today, many having displaced their ancestors. New and important languages are always on the horizon, and developers find themselves encountering and learning new languages every year. Yet none of these languages is ideal for every application.

In every domain where computer programs run, there is an opportunity to bridge the gap between humans and computers. Humans work with text, and computers work with objects. By learning to write parsers, you learn to bridge the gap between computers and the users of your language. You can create a new language for any niche, defining how your users interact with computers using text.

1.1 The Role of Parsers

English is a powerful language. With it, we can write plays and sonnets, grocery lists and business plans, love notes and contracts. A self-evident example is this book, which uses the English language to explain how to write parsers using the Java computer language. If computers could understand English, we would have little need for Java or any other programming language. Perhaps there is some potential to invent a human language that is less flexible and less ambiguous than English—for critical tasks such as launching a spacecraft—but people generally thrive using natural languages, applying the flexibility and even the ambiguity of language to good purpose. With computers, however, this does not work because they require precise commands to execute correctly.

Computers understand very little, and arguably nothing at all. They can add numbers and move strings of text but cannot in themselves understand the idea of, say, doing something 10 times. So on the one hand we have English, which is enormously expressive, and on the other hand we have computers, which understand almost nothing. This is why programming languages emerge. A programming language

such as Java is a compromise between the expressive eloquence of English and the primitive receptive abilities of a computer. Most of the words in Java are English words, and these words typically retain their English meaning. For example, `while` in Java has essentially the same meaning as the word *while* in an English sentence. Other words (`public`, for example) have meaning that is specific to a programming concept but is still related to the same word in English. Java is as English-like as it can be, but Java focuses on its task of enabling us to command computers.

It is reasonable to ask what the ideal programming language is, and indeed this question leads to progress and new languages such as Java. You might agree that Java is a better language because it is *object-oriented*. People tend to mentally model the world in terms of objects, and Java's object orientation makes it a ready receptacle for these thoughts. It simplifies the connections between, say, a real furnace and a `Furnace` object, and that in turn simplifies the control of a real furnace from a Java program. Object-oriented languages ease the programmer's task of interacting with objects in the real world. In addition to its object orientation, Java has many other features that make it the right choice of a computer language for many applications.

Java is an excellent choice of language when the problem at hand requires giving computers precise commands. However, if you want to describe what a Web page should look like, HTML (Hypertext Markup Language) and XML (Extensible Markup Language) are more appropriate choices. XML is different from Java—a different compromise between human understanding and computer understanding.

The ideal compromise between English and computer languages depends on the context and purpose of what you are doing with the computer. This is why every programmer can benefit from learning how to create new languages. The point of learning to write parsers is that you can craft a language that fits the context you are working in. You can write a search language that is easier to use than SQL (Structured Query Language) and more specific to the data in your database. You can write a language that lets you command a robot, or one that lets you move an order through its workflow. You can create a privilege language for logically modeling which users should have access to which transactions in a system. Whatever your application is, parsers let you concoct an English-like interface to it.

Parsers help computers, which work with objects, to cooperate with people, who read and write text. In practice, particularly in Java-based parsers, this implies that parsers translate text into objects. For example, a parser can translate a textual command for a robot into a command object. Another parser might translate a textual description of a product into an object that represents the product. A query parser translates textual queries into commands that a query engine understands. Parsers, then, translate between text and objects, letting you trade text for objects and objects for text, in a way that exactly suits the domain you are working in.

As a language designer, you will craft a compromise between a language that is easy for humans to use and a language that computers can understand. It is well worth learning to write parsers because you will have many chances in your career to orchestrate how humans and computers interact using text.

1.2 What Is a Language?

For the purposes of this book, a *language* is a set of strings. For example,

```
{"Hello, World."}
```

is a very little language. Another little example is the following:

```
{"", "a", "aa", "aaa", ...}
```

This language is the set of strings of zero or more *a*s. The description of this set uses "..." to mean that the initial pattern continues. The strings that make up a language are the *elements* of that language. A *parser* is an object that recognizes the elements of a language. Most of the parsers in this book also build an object as the result of recognizing a language element.

Interesting languages usually contain strings that follow a certain pattern and are related in some way. For example, the following well-known languages are the subject of many books:

- Structured Query Language (SQL)
- Hypertext Markup Language (HTML)
- Extensible Markup Language (XML)

You can think of these languages as specifying patterns of admissible text. In terms of *sets*, SQL is the language that contains all valid SQL strings. Similarly, Java as a language is the set of strings that are valid Java classes and interfaces. Every string either defines or does not define a valid Java class.

Java is different from these three languages in that it is a *programming* language—a language that is geared toward execution on a computer. In addition to Java, popular examples include Basic, C, C++, and Smalltalk.

Famous languages tend to be large, difficult to implement, and subject to standardization and control. These features of famous languages can obscure how effective little languages can be. Unlike famous languages, little languages tend to be small, easy to implement, and subject to your own control.

1.3 The Organization of This Book

This book explains how to write parsers for new computer languages that you create. Each chapter focuses on background, techniques, or applications. Chapters on background give you the tools to build parsers. Chapters on techniques show you how to apply the tools. Chapters on applications explain how to create a specific parser for a particular type of language. Figure 1.1 shows the role of each chapter.

The structure of the book is fairly linear, with each chapter dependent only on preceding chapters. For example, Chapter 5, "Parsing Data Languages," depends primarily on background and techniques from Chapters 2 and 3. You can skip the middle chapters of the book—on tokenizing, mechanics, and new types—if you want to get right to the chapters on advanced languages.

		Background			
			Techniques		
	Chapter			Applications	
1	Introduction	■			
2	The Elements of a Parser	■			
3	Building a Parser		■		
4	Testing a Parser		■		
5	Parsing Data Languages			■	
6	Transforming a Grammar		■		
7	Parsing Arithmetic			■	
8	Parsing Regular Expressions			■	
9	Advanced Tokenizing		■		
10	Matching Mechanics	■			
11	Extending the Parser Toolkit		■		
12	Engines	■			
13	Logic Programming	■			
14	Parsing a Logic Language			■	
15	Parsing a Query Language			■	
16	Parsing an Imperative Language			■	
17	Directions			■	
A	UML Twice Distilled	■			

Figure 1.1 Each chapter in this book focuses on either background that supports later chapters, techniques that apply across parsers, or applications of parsers to a specific language type.

Chapter 2, "The Elements of a Parser," explains what a parser is, introduces the building blocks of applied parsers, and shows how to compose new parsers from existing ones.

Chapter 3, "Building a Parser," explains the steps in designing and coding a working parser.

Chapter 4, "Testing a Parser," explains how to test the features of a new language and how to use random testing to detect ambiguity and other potential problems.

Chapter 5, "Parsing Data Languages," shows how to create a parser to read elements of a data language. A *data language* is a set of strings that describe objects following a local convention. This chapter also explains that, given the opportunity, you should consider migrating data-oriented languages to XML.

Chapter 6, "Transforming a Grammar," explains how to ensure the correct behavior of operators in a language and how to avoid looping in a parser, which can follow from loops in a grammar.

Chapter 7, "Parsing Arithmetic," develops an arithmetic parser. Arithmetic usually appears as part of a larger language, and the ideas in this chapter reappear in the chapters on query, logic, and imperative languages. To focus on the correct interpretation of arithmetic, this chapter develops an independent parser.

Chapter 8, "Parsing Regular Expressions," develops a *regular expression* parser. A regular expression is a string that uses symbols to describe a pattern of characters. For example, `"~.txt"` might represent all file names that end with `.txt`. This chapter explains how to read a string such as `"~.txt"` and create a parser that will recognize all the strings the given pattern describes.

Chapter 9, "Advanced Tokenizing," describes the tokenizers that come with Java and the customizable tokenizer used in this book. *Tokenizing* a string means breaking the string into logical nuggets, something that lets you define a parser in terms of the nuggets instead of individual characters. The default operation of the tokenizer used in this book is sufficient for many languages, so customizing a tokenizer is an advanced topic.

Chapter 10, "Matching Mechanics," explains how the fundamental types of parsers in this book match text.

Chapter 11, "Extending the Parser Toolkit," explains how to extend a parser toolkit, introducing new types of terminals or completely new parser types.

Chapter 12, “Engines,” introduces a logic engine used in later chapters to build a logic language and a query language.

Chapter 13, “Logic Programming,” explains how to program in logic, which means programming with facts and rules.

Chapter 14, “Parsing a Logic Language,” explains how to construct a parser for a logic language. Chapter 13 explains logic programming, giving examples in the Logikus programming language; Chapter 14 explains how to construct a Logikus parser.

Chapter 15, “Parsing a Query Language,” describes how to construct a parser for a query language. A query language parser translates textual queries into calls to an engine. The engine proves the query against a source of rules and data and returns successful proofs as the result of the query.

Chapter 16, “Parsing an Imperative Language,” shows how to create a parser for an imperative language. An imperative language parser translates a textual script into a composition of commands that direct a sequence of actions.

Chapter 17, “Directions,” points out areas for further reading and programming.

Appendix A, “UML Twice Distilled,” explains the features of the Unified Modeling Language that this book applies.

1.4 Summary

Parsers strike a compromise between people and computers. The language recognized by a parser may be a simple data language, or it may be a programming language such as a query, logic, and imperative language. This book shows how to create parsers for all these languages so that you can fit just the right language into whatever niche you are programming for.