Picturing Programs
An introduction to computer programming

Stephen Bloch[1]

[1]Math/CS Department, Adelphi University. Supported in part by NSF grant 0618543. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Dedicated to my wife Deborah, with whom I would have done more fun things in the past year if I hadn't been busy writing a book.

Contents

Chapter 0

Introduction

0.1 Languages and dialects

Computers don't naturally understand human languages such as English. Instead, we invent artificial languages to communicate with them. These artificial languages are typically much simpler than any human language, so it's easier to learn them than for, say, an English speaker to learn Chinese. But it's still hard work. As with any language, you'll need to learn the spelling, punctuation, grammar, vocabulary, and idioms[1] of the new language.

Among the artificial languages people use to communicate with computers (and computers use to communicate with one another) are HTML, XML, SQL, Javascript, Java, C++, Python, Scheme, PHP, Ruby, sh, awk, Racket, and hundreds more. Some of these languages are called *programming* languages because they are used mostly to write programs — that is, to teach a computer new tricks by combining the tricks it already knows.

This is a book about how to write computer programs. Pretty much every such book chooses one particular programming language. I've chosen to use a new language called Racket (which is based on a 30-year-old language named Scheme, which is based on a 50-year-old language named Lisp, which is based on an 80-year-old mathematical theory named lambda-calculus...). But it's not a Racket book; the Racket language is not the *goal*, but only a *means* towards the goal of knowing how to program.

Here's why: throughout the history of computers, the dominant languages have changed every five to ten years. (Fortran, Cobol, BASIC, PL/I, Pascal, C++, Java, Python, ...) No matter which of these languages you learn, it will probably become obsolete in a few years. If you plan to get a job as a computer programmer next month, then by all means study the language(s) used in industry right now. But if you plan to get a job programming several years from now, you'll have to learn a new language then anyway. The current school term will be better spent learning more long-lasting skills, habits, and principles: how to structure a program, what steps to take in developing a program, how to manage your time so you finish the program on time, *etc.* And if you

[1] "Idiom" means the way a particular language is *typically* used by those who use it heavily. For example, if I said "This book is more good than any other programming book," you would know what I meant, but you would also know I wasn't a native English-speaker; a native English speaker would say "This book is *better* than any other programming book." Every language, including computer programming languages, has its own idioms.

don't plan to be a professional programmer at all, then you don't need to learn this year's "hot" language at all; you need to learn the important principles of programming, in whatever language will "get out of the way" and let you learn them.

In fact, we won't even be using very much of the Racket language. The software we use, a program named DrRacket, provides several *dialects* of Racket, intended for different kinds of users. (By way of analogy, the United States and England use different dialects of English: most of the words are the same, but sometimes the same words mean completely different things in different countries. Furthermore, an elementary school student, an economist, and a sculptor may all use English, but they use it differently, and they may use the same word to mean different things.) The "Beginning Student" dialect, in which we'll start, doesn't allow you to do some things that are technically legal Racket, but which tend to confuse beginning programmers. If you really need to do these things, you can switch to a larger dialect with a few mouse-clicks.

In this book, there will be no "black magic": nothing that you need to memorize on faith that you'll eventually understand it. On the first day, you will see just enough language to do what you need on the first day. By the end of the term, you will see just enough language to do what you need in one term. Any language feature that doesn't help to teach an important programming principle doesn't belong in this book. Most programming languages, frankly, don't allow me to do that: in C++ or Java, for example, the very first program you write requires knowing dozens of language features that won't be fully explained for months. Racket allows me to postpone irrelevant language features, and concentrate on the important stuff.

Racket is also a much simpler, more consistent language than C++, Java, or Python, so it takes much less time to learn. This, too, allows you to concentrate on the important stuff, which is how to write a program.

Again, Racket is only a means to an end. If six months after taking this course you don't remember any Racket at all but can follow the steps of solving a problem, as explained in this book, the course has been a success.

0.2 Problems, programs, and program testing

A computer program that answered only one specific question, like

> add 3 and 4

wouldn't be very useful. Most computer programs are written to be *general*, in that a *single* program can answer any one of *many similar questions*:

- add 3 and 4
- add 19 and -5
- add 102379 and -897250987

etc. Somebody writes the program to add two numbers once and for all; later on, when you *run* the program, you provide specific values like 3 and 4, and the program produces the right answer for those values. Run it again with different values, and it should produce the right answer for the new values instead.

To take a more realistic example, a word processor program is written to handle whatever words you choose to write. When you run the program, you provide specific words — a grocery list, a letter to your grandmother, the next best-selling novel — and

the program responds by doing things like formatting them to fit on a page. Likewise, when you run a Web browser, you provide a specific URL for a page you want to look at; the browser program uses the network to retrieve specific words and pictures from that Web page, and then arranges these words and pictures on the screen. If you've done a lot of Web surfing, you've probably found an occasional page that showed up on the screen as nonsense; this probably means the page had some weird information that the browser wasn't written to handle correctly.

For a computer program to be considered "correct", it has to produce the right answer for *all possible* values it might be given to work on — even the weird ones. One of the important steps in writing a computer program is *testing* it to make sure it works correctly. However, since there are usually far too many possible values to test them all, we have to *choose test cases*, being careful to pick not only the easy cases but also the weird ones, so that if there's something our program doesn't handle correctly, we find out as soon as possible so we can fix it.

A program that hasn't been tested convincingly is worthless: nobody will (or should!) trust the answers it produces. Indeed, if you *tell* me you've tested the program, but don't provide me with what I need in order to test it myself, I may not trust either you *or* the program.

So one of the themes of this book will be "how to tell whether your program is correct." We'll discuss how and when to choose good test cases, as well as how to interpret patterns of correct and incorrect test cases to track down the source of the error.

0.3 Using DrRacket

This section doesn't cover any "big ideas", only the details of how to get DrRacket to work the way you need it to in this book. If you've already got DrRacket and the `picturing-programs` library installed, you can skip this section.

0.3.1 Getting DrRacket

If you haven't got the DrRacket program installed on your computer already (it usually has a red-white-and-blue icon, a circle with the Greek letter λ on it), you'll need to get it. You can download it for free, for Windows, Macintosh, and Linux, from `http://www.racket-lang.org`. This textbook assumes you have a version of DrRacket numbered 5.0.1 or higher.

0.3.2 Starting DrRacket

Once you've got DrRacket downloaded and installed, you should be able to run it by double-clicking the icon. It should open a window with a few buttons across the top, and two large panes. In the lower pane (the "Interactions Pane", where we'll be working at first) should be a welcome message like

```
Welcome to DrRacket, version 5.1.
Language: Beginning Student.
>
```

(Your version number and language may be different.)

The "> " prompt is where you'll type things.

0.3.3 Choosing languages

DrRacket provides a number of different computer languages, most of which are dialects of Racket. For now, we want to be working in the "Beginning Student" language. If the welcome message says something other than "Beginning Student" (or perhaps "Beginning Student custom") after the word "Language:", do the following:

1. Pull down the "Language" menu and select "Choose Language..."
2. Find the group of languages named "How to Design Programs"
3. If necessary, click the triangle to the left of "How to Design Programs" to show its sub-headings
4. Select "Beginning Student"
5. Click "OK"
6. Quit DrRacket and start it again, and it should now say "Language: Beginning Student".

(You don't *really* have to quit and re-start DrRacket; you can get the same effect by clicking the "Run" button. However, quitting and restarting demonstrates that DrRacket remembers your choice of language from one time you use it to the next.)

0.3.4 Installing libraries

A "library", or "teachpack", is a collection of optional tools that can be added into DrRacket. For most of this book, we'll need one named `picturing-programs`.

Skip this section if you have DrRacket version 5.1 or later: `picturing-programs` is already installed on your computer.

If you don't already have the `picturing-programs` library, here's how to get it. You'll only have to do this once on any given computer.

1. Make sure your computer is connected to the Internet.
2. Start DrRacket.
3. From the "Language" menu, "Choose Language", then select "Use the language declared in the source".
4. Click "Run".
5. At the "> " prompt in the bottom half of the screen, type

   ```
   (require (planet sbloch/picturing-programs:2))
   ```

   exactly like that, with the parentheses and the slash and all. It may take a few seconds to a few minutes (most of which is updating the help system to include information on this library), but eventually you should see the message "Wrote file "picturing-programs.ss" to installed-teachpacks directory."
6. From the "Language" menu, "Choose Language", then click on to "How to Design Programs", then select "Beginning Student". Hit "Run" again.

0.3.5 Getting help

If you want to look up reference information about this library (or anything else in the language),

1. from the "Help" menu, choose "Help Desk".
2. find the search box at the top of the screen and type the name of a library or function you want to learn about. Then hit ENTER.
3. If the name is found, you'll get a list of places it appeared in the documentation. Click one of them (probably one that says it's from the "picturing-programs" library).
4. Documentation for that library or function should appear on the screen.

0.4 Textbook web site

In order to keep the cost of this book down, we've put all the illustrations in black and white. You can find colored versions of many of them, as well as corrections, updates, additions, image files, and downloadable versions of worked exercises (so you don't have to type them in by hand), *etc.* at `http://www.picturingprograms.com`.

Index

Bibliography

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, 1996.

[FFF+08a] Matthias Felleisen, Robert Bruce Findler, Kathi Fisler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Worlds: Imaginative Programming in DrScheme*. self-published on Web, http://world.cs.brown.edu, 2008.

[FFF+08b] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Kathryn Gray, Shriram Krishnamurthi, and Viera K. Proulx. How to design class hierarchies. In preparation, 2008.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: an Introduction to Programming and Computing*. MIT Press, 2001. See http://www.htdp.org.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, 15(12):1053–1058, Dec 1972.