Texts in Computer Science

Titles in this series now included in the Thomson Reuters Book Citation Index!

'Texts in Computer Science' (TCS) delivers high-quality instructional content for undergraduates and graduates in all areas of computing and information science, with a strong emphasis on core foundational and theoretical material but inclusive of some prominent applications-related content. TCS books should be reasonably self-contained and aim to provide students with modern and clear accounts of topics ranging across the computing curriculum. As a result, the books are ideal for semester courses or for individual self-study in cases where people need to expand their knowledge. All texts are authored by established experts in their fields, reviewed internally and by the series editors, and provide numerous examples, problems, and other pedagogical tools; many contain fully worked solutions.

The TCS series is comprised of high-quality, self-contained books that have broad and comprehensive coverage and are generally in hardback format and sometimes contain color. For undergraduate textbooks that are likely to be more brief and modular in their approach, require only black and white, and are under 275 pages, Springer offers the flexibly designed Undergraduate Topics in Computer Science series, to which we refer potential authors.

More information about this series at https://link.springer.com/bookseries/3191

Marco T. Morazán

Animated Problem Solving

An Introduction to Program Design Using Video Game Development

Marco T. Morazán
Department of Computer Science
Seton Hall University
South Orange, NJ, USA

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-030-85090-6 ISBN 978-3-030-85091-3 (eBook)
https://doi.org/10.1007/978-3-030-85091-3

To my parents, Doris and Marco, who taught me to love teaching and to realize that having an education is not a privilege but a responsibility.

Preface

Everybody engages in problem solving. It is a natural and inevitable part of life. Historically, the link between problem solving and programming has been less emphasized. When you write an essay, you are programming—at different levels many times. You make sure ideas flow and arguments make use of the data you are analyzing. You write several drafts of the essay. Each draft represents a refinement. Every paragraph has a point and you avoid repeating yourself. All this is part of programming, including computer programming. Programmers, people who solve problems using a computer, go through the exact same steps to write a program. The same steps are taken by a psychologist analyzing a patient and by a chemist experimenting in a laboratory. Even a painter engages in programming. No? Does a painter not want to elicit an outcome or an emotion in you? Indeed, how to achieve this is a problem that must be solved by the artist. Consider the painting *Sorrowing Old Man (At Eternity's Gate)* by Vincent van Gogh (search for it on the internet). Can you see the old man's sorrow? Can you imagine the weight of the years on him? If so, we can say that the painter successfully solved the problem. This brings us to another important component of problem solving: testing. It is not only important to solve a problem. It is equally important to test the solution to make sure it works and in many cases to make sure that the solution is efficient.

This book is about systematic problem solving or if you like about systematic reasoning. Unlike most textbooks about programming, this textbook is not about tinkering with or hacking code. This book is about making a plan to solve a problem and then implementing the solution. As we shall discover, it turns out that the solutions to many problems are similar. This should not come as a surprise because we often solve many problems using similar data. How do you do grocery shopping? You make a list of items and check them off as you put them in your cart. How do you manage your chores today? You make a list of chores and check them off as you get them done. Pretty similar, no? Similarities give rise to abstraction to avoid repetitions—or reinventing the proverbial wheel. This book, therefore, is also about abstraction. Thinking abstractly is a powerful tool in problem solving.

In this textbook, all the solutions to problems are expressed as programs. It is important to be somewhat precise about what a program is. A program is much more

than just code written using a programming language. Remember that a program is a solution to a problem. Therefore, a program has a design, code, examples of how it works, and tests. That is, it communicates how the problem is solved and illustrates that the solution works. If any of the mentioned components are missing, then we have an incomplete program. Would you believe someone who simply told you that $\texttt{n}^2$, where `n` is a nonnegative integer, is the sum of the first `n` odd numbers? Many readers would be skeptical. What if they also provided the following examples:

```
0^2 = 0
2^2 = 1 + 3
4^2 = 1 + 3 + 5 + 7
```

It is very likely that most readers would now feel more confident that the claim is true. It is the same in programming. We cannot simply say that here is a function that does this or that. We need to explain how the function computes its value, and we need to have examples that show how it works. The steps taken to design a program in a systematic manner is called a *design recipe*. In this textbook, you shall study many different design recipes. Each design recipe shall become a tool in your problem-solving toolbox.

There are two problem-solving techniques that are emphasized throughout the book: *divide and conquer* and *iterative refinement*. Divide and conquer is the process by which a large problem is broken into two or more smaller problems that are easier to solve and then the solutions for the smaller pieces are combined to create an answer to the problem. Iterative refinement is the process by which a solution to a problem is gradually made better—like the drafts of an essay. Mastering these techniques is essential to becoming a good problem solver and programmer.

Finally, problem solving ought to be fun. To this end, this book promises that by the end of it you will have designed and implemented a multiplayer video game that you can play with your friends over the internet. To achieve this, however, there is a lot about problem solving and programming that you must first learn. The game is developed using iterative refinement. As we learn about programming, we shall apply our new knowledge to develop increasingly better versions of the video game. In fact, every skill you develop for problem solving and program design is transferable to other (non-programming) domains and to other programming languages.

1 The Languages and the Parts of the Book

The book uses the `Racket` student languages to write programs. These languages are chosen for several reasons. The first is that they have an error-messaging system specifically designed for beginners. This means that unlike common programming languages the error messages are likely to make sense to beginners. If you do not understand an error message, do not hesitate to ask your professor or search for help online. The second is that the syntax is simple and easy to understand. This is important because the emphasis is always on problem solving and not on how

to correctly write expressions. The third is that the student languages progressively become richer. At the beginning, you have fewer features at your disposal and, therefore, the possible errors are fewer. The fourth reason is that the student languages come with powerful libraries to create graphics, animations, and video games. These libraries allow students to inject their own personalities in the development of games and animations. You are strongly encouraged to be creative. Finally, the fifth reason is that the `Racket` student languages are likely to put all students on the same playing field. Most students will be learning the syntax of the programming language together for the first time.

The book is divided into five parts. Part I focuses on the basics. It starts with how to write expressions. Once expressions are mastered, the first abstraction lesson introduces us to functions. In addition, this part introduces you to conditional expressions that allow you to write programs that make decisions. Just this much knowledge allows us to write interactive programs and puts us on our way to a multiplayer video game. As you shall discover, decision-making is fundamental to solving problems that involve information that has many varieties. For example, the whole numbers may be positive or negative—two varieties—and how a whole number is processed depends on which variety a given number belongs to. Think about how to compute the absolute value of a whole number.

Part II introduces you to compound data of finite size. Compound data has multiple values associated. For example, a point on the Cartesian plane is compound data of finite size. There are two values: an `x` coordinate and a `y` coordinate. Being able to define compound data of finite size to represent elements in the real or an imaginary world is a powerful skill to develop.

Part III introduces you to compound data of arbitrary size. This is data that has multiple values, but the number of values is not fixed. Once again, think about a grocery list. Sometimes there are no items in the list and at other times there may be 10, 6, or 17 items in the list. This is where you are introduced to *structural recursion*—a powerful data-processing strategy that uses divide and conquer to process data whose size is not fixed. The types of data that are introduced are lists, intervals, natural numbers, and binary trees. The knowledge developed is used to develop a video game that is more challenging for the player.

Part IV delves into abstraction. This section is where we learn how to eliminate repetitions in our solutions to problems. In fact, we learn how different data can be processed and different problems can be solved in exactly the same way. You are introduced to `generic programming`, which is abstraction over the type of data processed. This leads to the realization that functions are data and, perhaps more surprising, that data are functions. In other words, the line between data and functions is artificial—a fact that is not emphasized enough in Computer Science textbooks. This realization naturally leads to object-oriented programming—a topic that you are likely to study extensively.

Part V introduces you to distributed programming—using multiple computers to solve a problem. This is a topic that until now has never been addressed in a textbook for beginning programmers. The fact that you develop proficiency in program design makes it possible for this topic, common in modern computer applications, to be

discussed. If you have ever sent a text message or have ever played a game online, then you have benefitted from and have used a distributed program. It is impossible, of course, to discuss all the nuances of distributed programming in this textbook. Nonetheless, you are introduced to a modern trend that is likely to be common throughout your professional career and beyond.

2 Acknowledgments

This book is the product of over ten years of work at Seton Hall University building on the shoulders of giants in Computer Science. There are many persons and groups who deserve credit for informing my work. The `Racket` community has been unequivocal in its support for the techniques that I have developed. There is an unpayable debt of gratitude owed to Matthias Felleisen from Northeastern University for our discussions over the years about Computer Science education, Liberal Arts education, and program design. My students and I have greatly benefitted from his support. Other Racketeers who have deeply influenced me are Shriram Krishnamurthi, Matthew Flatt, Robert Bruce Findler, and Kathi Fisler. This textbook is a tribute to our debates and their published work.

I would also like to thank the Trends in Functional Programming (TFP) and the Trends in Functional Programming in Education (TFPIE) communities. These communities provided (and continue to provide) a venue to discuss and present work advancing Computer Science education. I am grateful to many individuals including Peter Achten, Jurriaan Hage, Pieter Koopman, Simon Thompson, and Marko van Eekelen. Their insightful feedback has informed much of the material in this textbook.

Finally, I would like to thank Seton Hall University and its Department of Computer Science for supporting the development of the work presented in this textbook. In particular, the support of John T. Saccoman, Manfred Minimair, and Daniel Gross is appreciated. Most of all, I am grateful to all my CS1 students over the past decade who have informed my Computer Science education efforts. It is likely true that my students have learned a great deal in my courses, but it is an absolute certainty that I have learned more from them. They have refined the delivery of every idea found in this textbook. I am especially grateful to all my undergraduate tutors and teaching assistants, including Shamil Dzhatdoyev, Josie Des Rosiers, Nicholas Olson, Nicholas Nelson, Lindsey Reams, Craig Pelling, Barbara Mucha, Joshua Schappel, Sachin Mahashabde, Rositsa Abrasheva, Isabella Felix, and Sena Karsavran. Without my dedicated students at Seton Hall University and their insight into what students understood, this textbook would have been impossible.

Contents