

# Preface

About 12 years ago a student<sup>1</sup> asked me after one of my lectures in *Computability and Complexity* why he had to write tedious Turing machine programs, given that everyone programmed in languages like Java. He had a point. What is the best way to teach a *Computability and Complexity* module in the twenty-first century to a cohort of students who are used to programming in high level languages with modern tools and libraries; to students who live in a world of smart phones and 24h connectivity, and, even more importantly maybe, who have not been exposed to very much formal reasoning and mathematics?

Turn the clock back only two or three decades. Then, a first year in a computer science Bachelor degree mainly consisted of mathematics (analysis and linear algebra, later discrete maths, numerical analysis, and basic probability theory). When computability and complexity was taught in the second or third year, students were already acquainted with a formal and very mathematical language of discourse, maybe because computer science lecturers in the 80s were usually mathematicians by trade. Things have changed significantly. Curriculum designers for Bachelor degrees are under pressure to push more and more new exciting material into a three-year degree program that should prepare students for their lives as working IT professionals. Any new module moved into the curriculum necessarily forces another one out. Often, allegedly “unpopular” modules, including formal theory and mathematics, are the victims. As a consequence, computer science students have to a degree lost the skills to digest material presented in an extremely formal and symbolic fashion while, at the same time, they are prolific programmers and quite knowledgeable in the use of tools.

So, *Computability and Complexity*, do they really have to be taught using Turing machines or  $\mu$ -recursive functions? Do they have to be presented in the style a logician or mathematician would prefer? Seeking for alternatives, I eventually stumbled across Neil Jones’ fantastic book *Computability and Complexity—From a Programming Perspective*. The subtitle already gives away the book’s philosophy.

---

<sup>1</sup>Alexis Petrounias.

The leitmotif of Neil’s textbook is to present the most important results in computability and complexity theory “using programming techniques and motivated by programming language theory” as well as “using a novel model of computation, differing from traditional ones in crucial aspects”.<sup>2</sup> The latter, *WHILE*, is a simple imperative language with one datatype of lists (s-expressions) à la LISP. Admittedly, this language is not Java, but it has the hallmarks of a modern high level language and is infinitely more comfortable to program in than Turing machines or Gödel numbers. Java, or any similar powerful language, would be impractical for our purposes “since proofs about them would be too complex to be easily understood”.<sup>3</sup>

So when rebranding the module under the name *Limits of Computation* in 2008, I adopted Neil’s book as course textbook. Delivering an introductory, one semester final-year module, I picked the most important and appealing chapters. This was easy as the design of the book was exactly made to mix and match.<sup>4</sup> Soon, however, it turned out that students found Neil’s book tough going. In fact, this became more apparent as the years went past. There were several factors. First of all, Neil’s students would have had ML, a functional language with built in list type, as a first programming language, whereas our students were raised on Java. Yet, the datatype of *WHILE* is a functional one, and this caused more problems to the students than anticipated. Second, and more importantly, I had not put enough attention to the prerequisites. Neil expected readers of his book to be senior undergraduate students “with good mathematical maturity.”<sup>5</sup> It turned out that not all the third-year students had this maturity (given the heterogeneity of backgrounds and reduction of maths teaching in the undergraduate years one and two).

As a response to mitigate the issues above, I started writing explicit notes to accompany my slides, intended as additional comments and explanations for the selected book chapters. I ended up adding more and more new material and rearranging it. The results of this effort are the 23 individual lectures of this book. A (British) semester is 12 weeks long, which usually requires 24 lectures to be delivered. The shortfall of one lecture is intentional, it acts as a buffer (in case things take more time) and also allows for extra events like invited talks or in-class tests.

This book was heavily influenced by Neil Jones’s textbook, which is clearly visible in some chapters. To pay homage to his book, its telling subtitle “From a Programming Perspective” has been adopted.

Brighton  
November 2015

Bernhard Reus

---

<sup>2</sup>Preface of Neil’s book, page x.

<sup>3</sup>The book “Understanding Computation From Simple Machines to Impossible Programs” by Tom Stuart, published by O’Reilly in 2013, appears to follow the same idea and philosophy, using Ruby as programming language. It does not deal with complexity however.

<sup>4</sup>Neil’s Preface, page xii.

<sup>5</sup>Neil’s Preface, page xiii.

Limits of Computation

From a Programming Perspective

Reus, B.

2016, XVIII, 348 p. 80 illus., Softcover

ISBN: 978-3-319-27887-2