Understanding Programming Languages

Cliff B. Jones

Understanding Programming Languages

Springer

Cliff B. Jones
School of Computing
Newcastle University
Newcastle upon Tyne, UK

ISBN 978-3-030-59256-1 ISBN 978-3-030-59257-8 (eBook)
https://doi.org/10.1007/978-3-030-59257-8

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The principal objective of this book is to teach a skill; to equip the reader with a way to understand programming languages at a deep level.

There exist far more programming languages than it makes sense even to attempt to enumerate. Very few of these languages can be considered to be free from issues that complicate –rather than ease– communication of ideas.

Designing a language is a non-trivial task and building tools to process the language requires a significant investment of time and resources. The formalism described in this book makes it possible to experiment with features of a programming language far more cheaply than by building a compiler. This makes it possible to think through combinations of language features and avoid unwanted interactions that can confuse users of the language. In general, engineers work long and hard on designs before they commit to create a physical artefact; software engineers need to embrace formal methods in order to avoid wasted effort.

The principal communication mode that humans use to make computers perform useful functions is to write programs — normally in "high-level" programming languages. The actual instruction sets of computers are low-level and constructing programs at that level is tedious and unintuitive (I say this from personal experience having even punched such instructions directly into binary cards). Furthermore these instruction sets vary widely so another bonus from programming in a language like Java is that the effort can migrate smoothly to computer architectures that did not even exist when the program was written.

General-purpose programming languages such as Java are referred to simply as "High-Level Languages" (HLLs). Languages for specific purposes are called "Domain Specific" (DSLs). HLLs facilitate expression of a programmer's intentions by abstracting away from details of particular machine architectures: iteration can be expressed in an HLL by an intuitive construct — entry and return from common code can be achieved by procedure calls or method invocation. Compilers for HLLs also free a programmer from worrying about when to use fast registers versus slower store accesses.

Designing an HLL is a challenging engineering task: the bigger the gap between its abstraction level and the target hardware architecture, the harder the task for the compiler designers. A large gap can also result in programmers complaining that they cannot get the same efficiency writing in the HLL as if they were to descend to the machine level.

An amazing number of HLLs have been devised. There are many concepts that recur in different languages but often deep similarities are disguised by arbitrary syntactic differences. Sadly, combinations of known concepts with novel ideas often interact badly and create hidden traps for users of the languages (both writers and readers).

Fortunately, there is a less expensive way of sorting out the meaning of a programming language than writing a compiler. This book is about describing the meaning (semantics) of programming languages. A major objective is to teach the skill of writing semantic descriptions because this provides a way to think out and make choices about the semantic features of a programming language in a cost-effective way. In one sense a compiler (or an interpreter) offers a complete formal description of the semantics of its source language. But it is not something that can be used as a basis for reasoning about the source language; nor can it serve as a definition of a programming language itself since this must allow a range of implementations. Writing a formal semantics of a language can yield a far shorter description and one about which it is possible to reason. To think that it is a sensible engineering process to go from a collection of sample programs directly to coding a compiler would be naive in the extreme. What a formal semantic description offers is a way to think out, record and analyse design choices in a language; such a description can also be the basis of a systematic development process for subsequent compilers. To record a description of the semantics of a language requires a notation — a "meta-language". The meta-language used in this book is simple and is covered in easy steps throughout the early chapters.

The practical approach adopted throughout this book is to consider a list of issues that arise in extant programming languages. Although there are over 60 such issues mentioned in this book, there is no claim that the list is exhaustive; the issues are chosen to throw up the challenges that their description represents. This identifies a far smaller list of techniques that must be mastered in order to write formal semantic descriptions. It is these techniques that are the main takeaway of the current book.

Largely in industry (mainly in IBM), I have worked on formal semantic descriptions since the 1960s[1] and have taught the subject in two UK universities. The payoff of being able to write formal abstract descriptions of programming languages is that this skill has a far longer half-life than programming languages that come and go: one can write a description of any language that one wants to understand; a language designer can experiment with combinations of ideas and eliminate "feature interactions" at far less cost and time than would be the case with writing a compiler.

The skill that this book aims to communicate will equip the reader with a way to understand programming languages at a deep level. If the reader then wants to

[1] This included working with the early operational semantic descriptions of PL/I and writing the later denotational description of that language. PL/I is a huge language and, not surprisingly, contains many examples of what might be regarded as poor design decisions. These are often taken as cautionary tales in the book but other languages such as Ada or CHILL are not significantly better.

design a programming language (DSL or HLL), the skill can be put to use in creating a language with little risk of having hidden feature interactions that will complicate writing a compiler and/or confuse subsequent users of the language.

In fact, having mastered the skill of writing a formal semantic description, the reader should be able to sketch the state and environment of a formal model for most languages in a few pages. Communicating this practical skill is the main aim of this book; it seeks neither to explore theoretical details nor to teach readers how to build compilers.

Using this book

The reader is assumed to know at least one (imperative) HLL and to be aware of discrete maths notations such as those for logic and set theory — [MS13], for example, covers significantly more than is expected of the reader. On the whole, the current book is intended to be self-contained with respect to notation.

The material in this book has been used in final-year undergraduate teaching for over a decade; it has evolved and the current text is an almost complete rewrite. Apart from a course environment, it is hoped that the book will influence designers of programming languages. As indicated in Chapter 1, current languages offer many unfortunate feature interactions which make their use in building major computer systems both troublesome and unreliable. Programming languages offer the essential means of expression for programmers — as such they should be as clean and free from hidden traps as possible. The repeated message throughout this book is that it is far cheaper and more efficient to think out issues of language design before beginning to construct compilers or interpreters that might lock in incompletely thought-out design ideas.

Most chapters in the book offer projects, which vary widely in their challenge. They are not to be thought of as offering simple finger exercises — some of them ask for complete descriptions of languages — the projects are there to suggest what a reader might want to think about at that stage of study.

Some sections are starred as not being essential to the main argument; most chapters include a section of "further material". Both can be omitted on first reading.

Writing style

"The current author" normally eschews the first person (singular or plural) in technical writing; clearly, I have not followed this constraint in this preface. Some of the sections that close each chapter and occasional footnotes also use the first person singular when a particular observation warrants such employment.

Acknowledgements

I have had the pleasure of working with many colleagues and friends on the subject of programming language semantics. Rather than list them here, their names will crop up throughout the book. I have gained inspiration from students who have followed my courses at both Newcastle University and the University of Manchester. I'm extremely grateful to Jamie Charsley for his insertion of indexing commands. I owe a debt to Troy Astarte, Andrzej Blikle, Tom Helyer, Adrian Johnson and Jim Woodcock, who kindly offered comments on various drafts of this book. (All remaining errors are of course my responsibility.) My collaboration with Springer –especially with Ronan Nugent– has been a pleasure. I have received many grants from EPSRC over the years — specifically, the "Strata" Platform Grant helped support recent work on this book.

Contents