preface

A program is a description of an abstract, general solution to a specific problem. It is typically written in a formal language called a programming language. The primary purpose of a program is to be understood by fellow human beings, thereby spreading knowledge. In order to achieve maximal readability, a programming language should have certain properties:

1. It should be small and uniform;
2. It should be free from ambiguity;
3. It should provide a high degree of abstraction;
4. It should be independent from concrete computer architectures.

The first points are no-brainers. If a language is too complex or has no uniform syntax and semantics, programmers will have to look up things in the manual perpetually instead of concentrating on the actual problem. If the language introduces ambiguity, people will eventually choose one possible outcome internally and start writing programs that depend on their imagination instead of facts.

A high degree of abstraction means that the language should provide means of dealing with recurring tasks gracefully and without having to reinvent the wheel over and over again. This may seem like a contradiction to 1., but this text will show that this does not have to be the case.

A programming language that is used to describe algorithms in a readable way must be fully architecture-neutral. As soon as the language depends on features of a particular machine, the first principle is violated, because the knowledge that is necessary to understand a program then includes the knowledge of the underlying architecture.

The first part of this book introduces the concept of functional programming, describes a language that fulfills all of the above requirements, and shows how to solve simple problems in it.

Once a language has been chosen, it can be used to formulate solutions to all kinds of logic problems. Programming is limited to solutions of *logic* problems, because it cannot anwser questions like ‘‘how can we learn to live in peace?’’ and ‘‘why is life worth living?’’. These are the limits of science as we know it. The class of problems that *can* be solved by programs is much smaller. It typically involves very clearly defined tasks like

– find permutations of a set;
– find factors of an integer;
– represent an infinite sequence;
– translate formal language A to language B;
– find a pattern in a sequence of characters;
– solve a system of assertions.

Of course these tasks have to be defined in much greater detail in order to be able to solve them programmatically. This is what the second part of the book is about: creating general solutions to logic problems.

The language that will be used in this book is a minimalistic variant of Scheme called zenlisp. Its only data types are symbols and ordered pairs. Nothing else is required to describe algorithms for solving all of the logic problems listed above. The language may appear useless to you, because it does not provide any access to ‘‘real-world’’ application program interfaces and it cannot be used to write interactive programs, but I have chosen this language on purpose: it demonstrates that programming does not depend on these features.

The second part of the book shows how to apply the techniques of the first part to some problems of varying complexity. The topics discussed in this part range from simple functions for sorting or permuting lists to regular expression matching, formal language translation, and declarative programming. It contains the full source code to a source-to-source compiler, a meta-circular interpreter and a logic programming system.

The third part, finally, shows how to implement the abstraction layer that is necessary for solving problems in an abstract way on a concrete computer. It reproduces the complete and heavily annotated source code for an interpreter of zenlisp.

The first chapter of this part strives to deliver an example of readable code in a language that is not suitable for abstraction. It attempts to develop a programming style that does not depend on annotations to make the intention of the programmer clear. Comments are interspersed between functions, though, because prose is still easier to read than imperative code.

At this point the tour ends. It starts with an abstract and purely symbolic view on programming, advances to the application of symbolic programming to more complex problems, and concludes with the implementation of a symbolic programming system on actual computer systems.

I hope that you will enjoy the tour!

Nils M Holm, September 2008

contents