

# 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 answer 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 `zen!isp`. 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

## part one: symbolic programming . . . . . 9

1.	<b>basic aspects . . . . .</b>	<b>9</b>
1.1	symbols and variables . . . . .	9
1.2	functions . . . . .	10
1.2.1	calling functions . . . . .	11
1.2.2	function composition . . . . .	12
1.3	conditions . . . . .	13
1.4	recursion . . . . .	15
1.5	forms and expressions . . . . .	16
1.5.1	lists . . . . .	17
1.5.2	forms . . . . .	19
1.5.3	expressions . . . . .	20
1.6	recursion over lists . . . . .	20
2.	<b>more interesting aspects . . . . .</b>	<b>24</b>
2.1	variadic functions . . . . .	24
2.2	equality and identity . . . . .	26
2.2.1	comparing more complex structures . . . . .	27
2.3	more control . . . . .	29
2.4	structural recursion . . . . .	30
2.5	functions revisited . . . . .	33
2.5.1	bound and free variables . . . . .	34
2.6	local contexts . . . . .	35
2.6.1	closures . . . . .	36
2.6.2	recursive functions . . . . .	38
2.6.3	recursive closures . . . . .	39
2.6.4	recursive bindings . . . . .	40
2.7	higher-order functions . . . . .	42
2.7.1	mapping . . . . .	43
2.7.2	folding . . . . .	45
3.	<b>rather esoteric aspects . . . . .</b>	<b>47</b>
3.1	numeric functions . . . . .	47
3.1.1	numeric predicates . . . . .	48
3.1.2	integer functions . . . . .	50
3.1.3	rational functions . . . . .	51
3.1.4	type checking and conversion . . . . .	52
3.2	side effects . . . . .	54
3.2.1	subtle side effects . . . . .	55
3.2.2	evaluation . . . . .	56
3.3	metaprogramming . . . . .	56
3.3.1	programs hacking programs . . . . .	57

3.3.2	beta reduction by substitution . . . . .	59
3.4	packages . . . . .	62

**part two: algorithms . . . . . 63**

4.	<b>list functions . . . . .</b>	<b>63</b>
4.1	heads and tails . . . . .	63
4.2	find the n'th tail of a list . . . . .	64
4.3	count the atoms of a form . . . . .	64
4.4	flatten a tree . . . . .	65
4.5	partition a list . . . . .	66
4.6	folding over multiple lists . . . . .	67
4.7	substitute variables . . . . .	68
5.	<b>sorting . . . . .</b>	<b>69</b>
5.1	insertion sort . . . . .	69
5.2	quicksort . . . . .	70
5.3	mergesort . . . . .	72
5.4	unsorting lists . . . . .	73
6.	<b>logic and combinatoric functions . . . . .</b>	<b>78</b>
6.1	turning lists into sets . . . . .	78
6.2	union of sets . . . . .	78
6.3	find members with a given property . . . . .	79
6.4	verify properties . . . . .	80
6.5	combinations of sets . . . . .	80
6.6	permutations of sets . . . . .	83
7.	<b>math functions . . . . .</b>	<b>86</b>
7.1	sequences of numbers . . . . .	86
7.2	fast factorial function . . . . .	86
7.3	integer factorization . . . . .	88
7.4	partitioning integers . . . . .	89
7.5	exploring the limits of computability . . . . .	91
7.6	transposing matrixes . . . . .	93
8.	<b>data structures . . . . .</b>	<b>94</b>
8.1	generators . . . . .	94
8.2	streams . . . . .	95
8.3	ml-style records . . . . .	99
9.	<b>compilers . . . . .</b>	<b>106</b>
9.1	translating infix to prefix . . . . .	106
9.1.1	formal grammars . . . . .	106
9.1.2	left versus right recursion . . . . .	111
9.1.3	implementation . . . . .	113

9.2	translating prefix to infix . . . . .	117
9.3	regular expressions . . . . .	121
9.3.1	regular expression compilation . . . . .	123
9.3.2	regular expression matching . . . . .	126
9.4	meta-circular interpretation . . . . .	129
10.	<b>mexprc – an m-expression compiler . . . . .</b>	<b>139</b>
10.1	specification . . . . .	139
10.1.1	annotated grammar . . . . .	141
10.2	implementation . . . . .	143
10.2.1	lexical analysis . . . . .	144
10.2.2	syntax analysis and code synthesis . . . . .	147
10.3	example programs . . . . .	160
10.3.1	append lists . . . . .	161
10.3.2	the towers of hanoi . . . . .	162
10.3.4	n queens . . . . .	163
11.	<b>another micro kanren . . . . .</b>	<b>165</b>
11.1	introduction . . . . .	165
11.1.1	functions versus goals . . . . .	165
11.1.2	unification . . . . .	166
11.1.3	logic operators . . . . .	168
11.1.4	parameterized goals . . . . .	170
11.1.5	reification . . . . .	171
11.1.6	recursion . . . . .	171
11.1.7	converting predicates to goals . . . . .	173
11.1.8	converting functions to goals . . . . .	174
11.1.9	<b>cond</b> versus <b>any</b> . . . . .	175
11.1.10	first class variables . . . . .	176
11.1.11	first class goals . . . . .	178
11.1.12	negation . . . . .	179
11.1.13	cutting . . . . .	180
11.2	a logic puzzle . . . . .	182
11.3	implementation . . . . .	186
11.2.1	basics . . . . .	186
11.2.2	goals . . . . .	188
11.2.3	interface . . . . .	190
<b>part three: zenlisp implementation . . . . .</b>		<b>195</b>
12.	<b>c part . . . . .</b>	<b>195</b>
12.1	prelude and data declarations . . . . .	195
12.2	miscellaneous functions . . . . .	204
12.3	error reporting . . . . .	205
12.4	counting functions . . . . .	207
12.5	memory management . . . . .	208

12.6	symbol tables . . . . .	213
12.7	reader . . . . .	216
12.8	primitive operation handlers . . . . .	222
12.9	special form handlers . . . . .	229
12.10	evaluator . . . . .	247
12.11	printer . . . . .	255
12.12	initialization . . . . .	258
12.13	interpreter interface . . . . .	260
12.14	interpreter shell . . . . .	264
13.	<b>lisp part . . . . .</b>	<b>270</b>
13.1	base library . . . . .	270
13.2	iterator package . . . . .	274
13.3	natural math functions . . . . .	274
13.4	integer math functions . . . . .	286
13.5	rational math functions . . . . .	293

**appendix . . . . . 303**

A.1	tail call rules . . . . .	303
A.2	zenlisp functions . . . . .	304
A.2.1	definitions . . . . .	304
A.2.2	control . . . . .	305
A.2.3	lists . . . . .	305
A.2.4	miscellanea . . . . .	306
A.2.5	packages . . . . .	307
A.2.6	meta functions . . . . .	307
A.3	math functions . . . . .	308
A.4	working with zenlisp . . . . .	309
A.4.1	the development cycle . . . . .	311
A.5	zenlisp for the experienced schemer . . . . .	314
A.6	answers to some questions . . . . .	314
A.7	list of figures . . . . .	324
A.8	list of example programs . . . . .	324
A.9	code license . . . . .	326

**index . . . . . 327**