

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Our Philosophy . . . . .	7
1.2	The Structure of This Book . . . . .	7
1.3	The Language of This Book . . . . .	7
<b>2</b>	<b>Everything (We Will Say) About Parsing</b>	<b>10</b>
2.1	A Lightweight, Built-In First Half of a Parser . . . . .	10
2.2	A Convenient Shortcut . . . . .	10
2.3	Types for Parsing . . . . .	11
2.4	Completing the Parser . . . . .	12
2.5	Coda . . . . .	13
<b>3</b>	<b>A First Look at Interpretation</b>	<b>13</b>
3.1	Representing Arithmetic . . . . .	14
3.2	Writing an Interpreter . . . . .	14
3.3	Did You Notice? . . . . .	15
3.4	Growing the Language . . . . .	16
<b>4</b>	<b>A First Taste of Desugaring</b>	<b>16</b>
4.1	Extension: Binary Subtraction . . . . .	17
4.2	Extension: Unary Negation . . . . .	18
<b>5</b>	<b>Adding Functions to the Language</b>	<b>19</b>
5.1	Defining Data Representations . . . . .	19
5.2	Growing the Interpreter . . . . .	21
5.3	Substitution . . . . .	22
5.4	The Interpreter, Resumed . . . . .	23
5.5	Oh Wait, There's More! . . . . .	25
<b>6</b>	<b>From Substitution to Environments</b>	<b>25</b>
6.1	Introducing the Environment . . . . .	26
6.2	Interpreting with Environments . . . . .	27
6.3	Deferring Correctly . . . . .	29
6.4	Scope . . . . .	30
6.4.1	How Bad Is It? . . . . .	30
6.4.2	The Top-Level Scope . . . . .	31
6.5	Exposing the Environment . . . . .	31
<b>7</b>	<b>Functions Anywhere</b>	<b>31</b>
7.1	Functions as Expressions and Values . . . . .	32
7.2	Nested What? . . . . .	35
7.3	Implementing Closures . . . . .	37
7.4	Substitution, Again . . . . .	38
7.5	Sugaring Over Anonymity . . . . .	39

<b>8</b>	<b>Mutation: Structures and Variables</b>	<b>41</b>
8.1	Mutable Structures . . . . .	41
8.1.1	A Simple Model of Mutable Structures . . . . .	41
8.1.2	Scaffolding . . . . .	42
8.1.3	Interaction with Closures . . . . .	43
8.1.4	Understanding the Interpretation of Boxes . . . . .	44
8.1.5	Can the Environment Help? . . . . .	46
8.1.6	Introducing the Store . . . . .	48
8.1.7	Interpreting Boxes . . . . .	49
8.1.8	The Bigger Picture . . . . .	54
8.2	Variables . . . . .	57
8.2.1	Terminology . . . . .	57
8.2.2	Syntax . . . . .	57
8.2.3	Interpreting Variables . . . . .	58
8.3	The Design of Stateful Language Operations . . . . .	59
8.4	Parameter Passing . . . . .	60
<b>9</b>	<b>Recursion and Cycles: Procedures and Data</b>	<b>62</b>
9.1	Recursive and Cyclic Data . . . . .	62
9.2	Recursive Functions . . . . .	64
9.3	Premature Observation . . . . .	65
9.4	Without Explicit State . . . . .	66
<b>10</b>	<b>Objects</b>	<b>67</b>
10.1	Objects Without Inheritance . . . . .	67
10.1.1	Objects in the Core . . . . .	68
10.1.2	Objects by Desugaring . . . . .	69
10.1.3	Objects as Named Collections . . . . .	69
10.1.4	Constructors . . . . .	70
10.1.5	State . . . . .	71
10.1.6	Private Members . . . . .	71
10.1.7	Static Members . . . . .	72
10.1.8	Objects with Self-Reference . . . . .	72
10.1.9	Dynamic Dispatch . . . . .	74
10.2	Member Access Design Space . . . . .	75
10.3	What (Goes In) Else? . . . . .	75
10.3.1	Classes . . . . .	76
10.3.2	Prototypes . . . . .	78
10.3.3	Multiple Inheritance . . . . .	78
10.3.4	Super-Duper! . . . . .	79
10.3.5	Mixins and Traits . . . . .	79

<b>11</b>	<b>Memory Management</b>	<b>81</b>
11.1	Garbage . . . . .	81
11.2	What is “Correct” Garbage Recovery? . . . . .	81
11.3	Manual Reclamation . . . . .	82
11.3.1	The Cost of Fully-Manual Reclamation . . . . .	82
11.3.2	Reference Counting . . . . .	83
11.4	Automated Reclamation, or Garbage Collection . . . . .	84
11.4.1	Overview . . . . .	84
11.4.2	Truth and Provability . . . . .	85
11.4.3	Central Assumptions . . . . .	85
11.5	Conservative Garbage Collection . . . . .	86
11.6	Precise Garbage Collection . . . . .	87
<b>12</b>	<b>Representation Decisions</b>	<b>87</b>
12.1	Changing Representations . . . . .	87
12.2	Errors . . . . .	89
12.3	Changing Meaning . . . . .	89
12.4	One More Example . . . . .	90
<b>13</b>	<b>Desugaring as a Language Feature</b>	<b>91</b>
13.1	A First Example . . . . .	91
13.2	Syntax Transformers as Functions . . . . .	93
13.3	Guards . . . . .	95
13.4	Or: A Simple Macro with Many Features . . . . .	95
13.4.1	A First Attempt . . . . .	95
13.4.2	Guarding Evaluation . . . . .	97
13.4.3	Hygiene . . . . .	98
13.5	Identifier Capture . . . . .	99
13.6	Influence on Compiler Design . . . . .	101
13.7	Desugaring in Other Languages . . . . .	101
<b>14</b>	<b>Control Operations</b>	<b>102</b>
14.1	Control on the Web . . . . .	102
14.1.1	Program Decomposition into Now and Later . . . . .	104
14.1.2	A Partial Solution . . . . .	104
14.1.3	Achieving Statelessness . . . . .	106
14.1.4	Interaction with State . . . . .	107
14.2	Continuation-Passing Style . . . . .	109
14.2.1	Implementation by Desugaring . . . . .	110
14.2.2	Converting the Example . . . . .	114
14.2.3	Implementation in the Core . . . . .	115
14.3	Generators . . . . .	117
14.3.1	Design Variations . . . . .	117
14.3.2	Implementing Generators . . . . .	119
14.4	Continuations and Stacks . . . . .	121
14.5	Tail Calls . . . . .	123

14.6	Continuations as a Language Feature . . . . .	124
14.6.1	Presentation in the Language . . . . .	125
14.6.2	Defining Generators . . . . .	126
14.6.3	Defining Threads . . . . .	127
14.6.4	Better Primitives for Web Programming . . . . .	131
<b>15</b>	<b>Checking Program Invariants Statically: Types</b>	<b>131</b>
15.1	Types as Static Disciplines . . . . .	133
15.2	A Classical View of Types . . . . .	134
15.2.1	A Simple Type Checker . . . . .	134
15.2.2	Type-Checking Conditionals . . . . .	139
15.2.3	Recursion in Code . . . . .	139
15.2.4	Recursion in Data . . . . .	142
15.2.5	Types, Time, and Space . . . . .	144
15.2.6	Types and Mutation . . . . .	146
15.2.7	The Central Theorem: Type Soundness . . . . .	147
15.3	Extensions to the Core . . . . .	148
15.3.1	Explicit Parametric Polymorphism . . . . .	148
15.3.2	Type Inference . . . . .	155
15.3.3	Union Types . . . . .	164
15.3.4	Nominal Versus Structural Systems . . . . .	170
15.3.5	Intersection Types . . . . .	171
15.3.6	Recursive Types . . . . .	172
15.3.7	Subtyping . . . . .	173
15.3.8	Object Types . . . . .	176
<b>16</b>	<b>Checking Program Invariants Dynamically: Contracts</b>	<b>179</b>
16.1	Contracts as Predicates . . . . .	181
16.2	Tags, Types, and Observations on Values . . . . .	182
16.3	Higher-Order Contracts . . . . .	183
16.4	Syntactic Convenience . . . . .	187
16.5	Extending to Compound Data Structures . . . . .	188
16.6	More on Contracts and Observations . . . . .	189
16.7	Contracts and Mutation . . . . .	189
16.8	Combining Contracts . . . . .	190
16.9	Blame . . . . .	191
<b>17</b>	<b>Alternate Application Semantics</b>	<b>195</b>
17.1	Lazy Application . . . . .	196
17.1.1	A Lazy Application Example . . . . .	196
17.1.2	What Are Values? . . . . .	197
17.1.3	What Causes Evaluation? . . . . .	198
17.1.4	An Interpreter . . . . .	199
17.1.5	Laziness and Mutation . . . . .	201
17.1.6	Caching Computation . . . . .	201
17.2	Reactive Application . . . . .	201

17.2.1	Motivating Example: A Timer . . . . .	202
17.2.2	Callback Types are Four-Letter Words . . . . .	203
17.2.3	The Alternative: Reactive Languages . . . . .	204
17.2.4	Implementing Transparent Reactivity . . . . .	205