## Contents

1 **Introduction**  
   1.1 Our Philosophy  
   1.2 The Structure of This Book  
   1.3 The Language of This Book  

2 **Everything (We Will Say) About Parsing**  
   2.1 A Lightweight, Built-In First Half of a Parser  
   2.2 A Convenient Shortcut  
   2.3 Types for Parsing  
   2.4 Completing the Parser  
   2.5 Coda  

3 **A First Look at Interpretation**  
   3.1 Representing Arithmetic  
   3.2 Writing an Interpreter  
   3.3 Did You Notice?  
   3.4 Growing the Language  

4 **A First Taste of Desugaring**  
   4.1 Extension: Binary Subtraction  
   4.2 Extension: Unary Negation  

5 **Adding Functions to the Language**  
   5.1 Defining Data Representations  
   5.2 Growing the Interpreter  
   5.3 Substitution  
   5.4 The Interpreter, Resumed  
   5.5 Oh Wait, There's More!  

6 **From Substitution to Environments**  
   6.1 Introducing the Environment  
   6.2 Interpreting with Environments  
   6.3 Deferring Correctly  
   6.4 Scope  
   6.4.1 How Bad Is It?  
   6.4.2 The Top-Level Scope  
   6.5 Exposing the Environment  

7 **Functions Anywhere**  
   7.1 Functions as Expressions and Values  
   7.2 Nested What?  
   7.3 Implementing Closures  
   7.4 Substitution, Again  
   7.5 Sugaring Over Anonymity
# Mutation: Structures and Variables

## 8.1 Mutable Structures
- 8.1.1 A Simple Model of Mutable Structures
- 8.1.2 Scaffolding
- 8.1.3 Interaction with Closures
- 8.1.4 Understanding the Interpretation of Boxes
- 8.1.5 Can the Environment Help?
- 8.1.6 Introducing the Store
- 8.1.7 Interpreting Boxes
- 8.1.8 The Bigger Picture

## 8.2 Variables
- 8.2.1 Terminology
- 8.2.2 Syntax
- 8.2.3 Interpreting Variables

## 8.3 The Design of Stateful Language Operations

## 8.4 Parameter Passing

# Recursion and Cycles: Procedures and Data

## 9.1 Recursive and Cyclic Data

## 9.2 Recursive Functions

## 9.3 Premature Observation

## 9.4 Without Explicit State

# Objects

## 10.1 Objects Without Inheritance
- 10.1.1 Objects in the Core
- 10.1.2 Objects by Desugaring
- 10.1.3 Objects as Named Collections
- 10.1.4 Constructors
- 10.1.5 State
- 10.1.6 Private Members
- 10.1.7 Static Members
- 10.1.8 Objects with Self-Reference
- 10.1.9 Dynamic Dispatch

## 10.2 Member Access Design Space

## 10.3 What (Goes In) Else?
- 10.3.1 Classes
- 10.3.2 Prototypes
- 10.3.3 Multiple Inheritance
- 10.3.4 Super-Duper!
- 10.3.5 Mixins and Traits
## Memory Management

### 11.1 Garbage

### 11.2 What is “Correct” Garbage Recovery?

### 11.3 Manual Reclamation

#### 11.3.1 The Cost of Fully-Manual Reclamation

#### 11.3.2 Reference Counting

### 11.4 Automated Reclamation, or Garbage Collection

#### 11.4.1 Overview

#### 11.4.2 Truth and Provability

#### 11.4.3 Central Assumptions

### 11.5 Conservative Garbage Collection

### 11.6 Precise Garbage Collection

## Representation Decisions

### 12.1 Changing Representations

### 12.2 Errors

### 12.3 Changing Meaning

### 12.4 One More Example

## Desugaring as a Language Feature

### 13.1 A First Example

### 13.2 Syntax Transformers as Functions

### 13.3 Guards

### 13.4 Or: A Simple Macro with Many Features

#### 13.4.1 A First Attempt

#### 13.4.2 Guarding Evaluation

#### 13.4.3 Hygiene

### 13.5 Identifier Capture

### 13.6 Influence on Compiler Design

### 13.7 Desugaring in Other Languages

## Control Operations

### 14.1 Control on the Web

#### 14.1.1 Program Decomposition into Now and Later

#### 14.1.2 A Partial Solution

#### 14.1.3 Achieving Statelessness

#### 14.1.4 Interaction with State

### 14.2 Continuation-Passing Style

#### 14.2.1 Implementation by Desugaring

#### 14.2.2 Converting the Example

#### 14.2.3 Implementation in the Core

### 14.3 Generators

#### 14.3.1 Design Variations

#### 14.3.2 Implementing Generators

### 14.4 Continuations and Stacks

### 14.5 Tail Calls
14.6 Continuations as a Language Feature

14.6.1 Presentation in the Language

14.6.2 Defining Generators

14.6.3 Defining Threads

14.6.4 Better Primitives for Web Programming

15 Checking Program Invariants Statically: Types

15.1 Types as Static Disciplines

15.2 A Classical View of Types

15.2.1 A Simple Type Checker

15.2.2 Type-Checking Conditionals

15.2.3 Recursion in Code

15.2.4 Recursion in Data

15.2.5 Types, Time, and Space

15.2.6 Types and Mutation

15.2.7 The Central Theorem: Type Soundness

15.3 Extensions to the Core

15.3.1 Explicit Parametric Polymorphism

15.3.2 Type Inference

15.3.3 Union Types

15.3.4 Nominal Versus Structural Systems

15.3.5 Intersection Types

15.3.6 Recursive Types

15.3.7 Subtyping

15.3.8 Object Types

16 Checking Program Invariants Dynamically: Contracts

16.1 Contracts as Predicates

16.2 Tags, Types, and Observations on Values

16.3 Higher-Order Contracts

16.4 Syntactic Convenience

16.5 Extending to Compound Data Structures

16.6 More on Contracts and Observations

16.7 Contracts and Mutation

16.8 Combining Contracts

16.9 Blame

17 Alternate Application Semantics

17.1 Lazy Application

17.1.1 A Lazy Application Example

17.1.2 What Are Values?

17.1.3 What Causes Evaluation?

17.1.4 An Interpreter

17.1.5 Laziness and Mutation

17.1.6 Caching Computation

17.2 Reactive Application
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2.1 Motivating Example: A Timer</td>
<td>202</td>
</tr>
<tr>
<td>17.2.2 Callback Types are Four-Letter Words</td>
<td>203</td>
</tr>
<tr>
<td>17.2.3 The Alternative: Reactive Languages</td>
<td>204</td>
</tr>
<tr>
<td>17.2.4 Implementing Transparent Reactivity</td>
<td>205</td>
</tr>
</tbody>
</table>