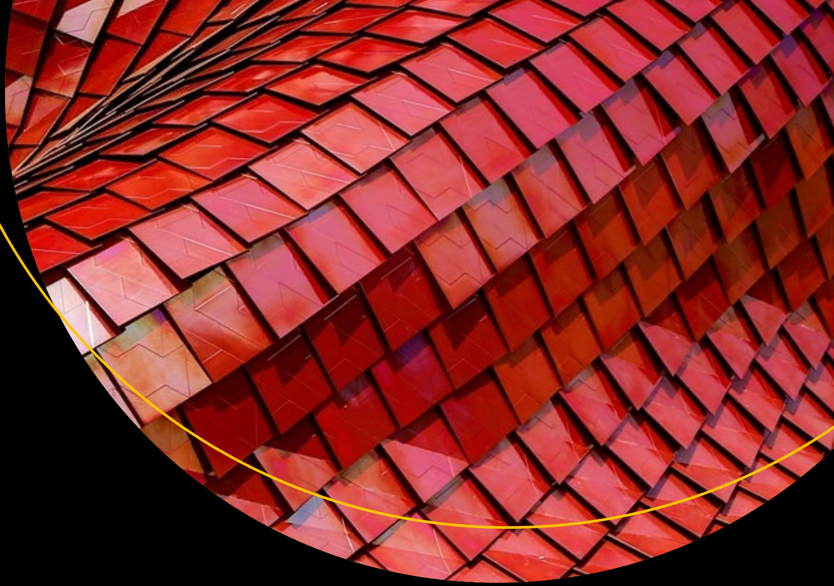

The Common Lisp Condition System

Beyond Exception Handling with Control Flow Mechanisms

—

Michał “phoe” Herda

Apress®

The Common Lisp Condition System

Beyond Exception Handling with Control Flow Mechanisms

Michał "phoe" Herda

Apress®

The Common Lisp Condition System: Beyond Exception Handling with Control Flow Mechanisms

Michał "phoe" Herda
Krakow, Poland

ISBN-13 (pbk): 978-1-4842-6133-0 ISBN-13 (electronic): 978-1-4842-6134-7
https://doi.org/10.1007/978-1-4842-6134-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Ricardo Gomez on Unsplash (www.unsplash.com)

Printed on acid-free paper

Table of Contents

About the Author ix

About the Technical Reviewers xi

Introduction xiii

Preface xvii

Chapter 1: Basic concepts 1

1.1 Dynamic variables 1

1.1.1 Dynamic variables in C 1

1.1.2 Dynamic variables in Common Lisp 8

1.1.3 Alternatives in other languages 13

1.2 Non-local transfers of control 14

1.2.1 TAGBODY and GO 16

1.2.2 BLOCK and RETURN-FROM/RETURN 18

1.2.3 CATCH and THROW 20

1.3 Lexical closures 21

Chapter 2: Introducing the condition system 25

2.1 A simple system of hooks 25

2.1.1 Hook #1: Launching Counter-Strike 27

2.1.2 Hook #2: Only call Counter-Strike players 33

2.1.3 Hook #3: Only call parents… maybe 34

2.1.4 Hook #4: Holiday wishes 35

2.1.5 Accumulating hooks 36

2.1.6 Hook #5: Calling Tom's girlfriend again 37

2.1.7 Multiple types of hooks 39

2.1.8 Summary: The hook subsystem 42

2.2 A simple system of condition handlers ... 43
2.2.1 Exception handling ... 50
2.2.2 Protection against transfers of control ... 58
2.2.3 Clustering ... 60
2.2.4 Summary: The handler subsystem ... 62
2.3 A simple system of choices ... 62
2.3.1 Kate and Mark ... 63
2.3.2 Choice #1: Escape ... 64
2.3.3 Choice #2: Excuses ... 72
2.3.4 Summary: the choice subsystem ... 74
2.4 A simple system of restarts ... 75
2.4.1 Interactive restarts ... 84
2.5 A simple system of actually restarting restarts ... 87
2.5.1 Restarts that perform a non-local exit ... 87
2.5.2 From RESTART-BIND to RESTART-CASE ... 90
2.5.3 Simple restarts ... 93
2.5.4 Standard restarts and restart-invoking functions ... 94
2.5.5 Defining custom restart-invoking functions ... 95
2.6 Reporting conditions and restarts ... 97
2.6.1 Printing vs. reporting ... 97
2.6.2 Custom condition reports ... 98
2.6.3 Custom restart reports ... 101
2.7 Warnings ... 104
2.7.1 Different ways of warning ... 105
2.7.2 Muffling warnings ... 106
2.8 Assertions ... 107
2.8.1 Simple assertions via ASSERT ... 107
2.8.2 Type checking via CHECK-TYPE ... 109
2.8.3 Case assertions ... 110
2.8.4 Correctable case assertions ... 110
2.8.5 Arguments for continuable errors ... 111

2.9 A simple debugger 112
2.9.1 Reporting the condition in the debugger 113
2.9.2 Reporting the condition type in the debugger 114
2.9.3 Reporting the restarts in the debugger 115
2.9.4 Choosing the restarts in the debugger 116
2.9.5 Installing a custom debugger 118
2.9.6 Recursive debugger 120
2.9.7 Adding a REPL to the debugger 124
2.9.8 Backtraces 127
2.9.9 Associating conditions with restarts 128

Chapter 3: Implementing the Common Lisp condition system 135

3.1 Package definition 136
3.2 Conditions 138
3.2.1 Base class for conditions 138
3.2.2 Defining new condition types 140
3.3 Coercing data to conditions 146
3.4 Restart basics 148
3.4.1 Restart class 148
3.4.2 Restart visibility and computing restarts 150
3.4.3 Invoking restarts 153
3.5 Binding restarts 155
3.6 Restart cases 158
3.6.1 First iteration: basics 158
3.6.2 Second iteration: Forms instead of a function 163
3.6.3 Third iteration: Managing the keyword differences 167
3.6.4 Fourth iteration: Associating conditions with restarts 169
3.6.5 Implementing simple restarts 174
3.7 System-defined restarts 175
3.8 Assertions 177
3.8.1 Case failures 177
3.8.2 Case utilities 178

3.8.3 Non-correctable case assertions.... 180
3.8.4 Correctable case assertions.... 182
3.8.5 General assertions.... 186
3.9 Signaling.... 190
3.10 Handlers.... 195
3.10.1 Binding handlers.... 195
3.10.2 Handler cases.... 196
3.10.3 Testing assertions.... 202
3.11 A featureful debugger.... 203
3.11.1 Debugger commands.... 203
3.11.2 Evaluating Lisp forms.... 205
3.11.3 Reporting and returning conditions.... 206
3.11.4 Listing and invoking restarts.... 208
3.11.5 Debugger help and REPL.... 211
3.11.6 Debugger interface.... 216
3.12 Finishing touches.... 217
3.12.1 Integration.... 217
3.12.2 Additional work.... 218
Chapter 4: Wrapping up.... 221
4.1 The purpose of the condition system.... 222
4.2 Binding vs. casing.... 223
4.3 Separation of concerns.... 224
4.4 Algebraic effects.... 225
4.5 Downsides of the condition system.... 226
4.5.1 Separation from CLOS.... 226
4.5.2 Dynamic extent of restart objects.... 227
4.5.3 Speed.... 228
4.5.4 Optimization settings.... 228
4.5.5 Introspection.... 228
4.5.6 Concrete condition types.... 229

4.5.7 Warning conditions and #'WARN ... 230
4.5.8 Implementing custom SIGNAL-like operators ... 230
4.5.9 Lack of functional interfaces to handlers and restarts ... 230
4.5.10 Smaller issues ... 231
4.5.11 Summary: Downsides of the Common Lisp condition system ... 232
4.6 Condition system in practice ... 233
4.6.1 Unit test library: Collecting results ... 233
4.6.2 Web framework: Sending results over the network ... 234
4.6.3 GUI applications: Interactive querying ... 235
4.6.4 Generating data: Python-like generators ... 235
Chapter 5: Appendixes ... 237
5.1 Appendix A: Implementation of dynamic variables in C ... 237
5.2 Appendix B: Additional utilities for working with Common Lisp conditions ... 242
5.2.1 CALL-WITH-HANDLER and CALL-WITH-RESTART ... 242
5.2.2 HANDLER-BIND* and HANDLER-CASE* ... 246
5.2.3 HANDLER-BIND-CASE ... 249
5.3 Appendix C: Lisp macros 101 ... 252
5.3.1 Basics of macro writing ... 253
5.3.2 Backquote ... 254
5.3.3 Symbol capture ... 255
5.3.4 Order of evaluation ... 258
5.3.5 Multiple evaluation ... 259
5.3.6 When not to write macros ... 260
5.3.7 Reference ... 260
5.4 Appendix D: Condition system reference ... 260
5.4.1 Restarts and related functions ... 260
5.4.2 Condition-restart association ... 264
5.4.3 Restart macros ... 265
5.4.4 Standard restarts ... 268
5.4.5 Defining and instantiating conditions ... 270

5.4.6 Assertions 274
5.4.7 Condition signaling 278
5.4.8 Handler macros 281
5.4.9 Condition types 283
5.4.10 Debugger invocation 290

Index 293

About the Author

Michał "phoe" Herda is a professional programmer who has contributed to multiple parts of the Common Lisp ecosystem: CL implementations, existing and widely used CL utilities, documentation, and some of the new library ideas that he slowly pushes forward and works on. This book is his first literary work — an attempt to create the tutorial on the condition system which had been missing even all these years after ANSI Common Lisp was standardized.

About the Technical Reviewers

Adlai Chandrasekhar is busy detangling the moral implications of having suffered the cruelly unusual punishment of performing quality control procedures on rocket targeting electronics that did not offer a condition system, despite the operators having studied Common Lisp before enlistment.

Dave Cooper is CTO at Genworks International and enjoys spending his time maintaining, extending, and supporting the Genworks GDL Knowledge-Based Engineering System and the Gendl Project which comprises its open source core (all based on Common Lisp language).

John Cowan's friends describe him as knowing at least something about almost everything, his enemies as knowing far too much about far too much.

Vsevolod Domkin, a.k.a. vseloved, is a Lisp aficionado and professional Common Lisp developer for more than 10 years, who is also a writer (author of the book *Programming Algorithms*) and occasional teacher of the courses on system programming, algorithms, and natural language processing.

Michael Fiano is a Common Lisp developer with 12 years of experience, who has made many contributions to the Common Lisp ecosystem, including pngload, and various game development software such as Virality Engine.

Marco Heisig is a passionate Lisp hacker, free software advocate, and researcher in the fields of high-performance computing, numerical mathematics, and compiler technology.

Jerome Onwunalu is a researcher in the fields of mathematical optimization and fluid flow modeling and is currently using Common Lisp for the development of a large-scale integrated software application.

Georgiy Tugai is a professional compiler developer who, having reviewed this book, now bemoans even more often the lack of a proper condition system, or object system for that matter, in the more mainstream languages he uses at work.

Introduction

This book is intended to be a tutorial for the Common Lisp condition system, teaching its functioning and presenting a range of example uses. It is aimed at beginning and intermediate Lisp programmers, as well as intermediate programmers of other programming languages. This book provides detailed information specifically about the CL condition system and its control flow mechanisms, so it is envisioned as a supplement to already existing material for studying Common Lisp (CL) as a language. The book also contains a description of an example ANSI-conforming implementation of the condition system.

What is a condition system? Introduction by Kent M. Pitman

There have been many attempts to declare the Lisp family of languages dead, and yet it continues on in many forms. There are many explanations for this, but an obvious one is that it still contains ideas and features that aren't fully appreciated outside the Lisp community, and so it continues as both a refuge and an idea factory.

Gradually, other languages see the light and these important features migrate to other languages. For example, the Lisp community used to be unusual for standing steadfastly by automatic memory management and garbage collection when many said it couldn't be trusted to be efficient or responsive. In the modern world, however, many languages now presume that automatic memory management is normal and natural, as if this had never been a controversy. So times change.

But proper condition handling is something which other languages still have not figured out that they need. Java's `try/catch` and Python's `try/except` have indeed shown that these languages appreciate the importance of representing exceptional situations as objects. However, in adopting these concepts, they have left out restarts—a key piece of the puzzle.

When you raise an exception in Python, or throw one in Java, you are still just performing an immediate and blind transfer of control to the innermost available handler. This leaves out the rich experience that Common Lisp offers to perform actual reasoning about where to return to.

The Common Lisp condition system disconnects the ability to return to a particular place in the program from the necessity to do so and adds the ability to "look before you leap." In other languages, if you create a possible place to return to, that is what will get used. There is no ability to say "If a certain kind of error happens, this might be a good place to return to, but I don't have a strong opinion ahead of time on whether or not it definitely will be the right place."

The Common Lisp condition system distinguishes among three different activities: describing a problem, describing a possible solution, and selecting the right solution for a given problem. In other languages, describing a possible solution is the same as selecting that solution, so the set of things you can describe is necessarily less expansive.

This matters, because in other languages such as Python or Java, by the time your program first notices a problem, it already will have "recovered" from it. The "except" or "catch" part of your "try" statement will have received control. There will have been no intervening time. To invoke the error handling process *is* to transfer control. By the time any further application code is running, a stack unwind already will have happened. The dynamic context of the problem will be gone, and with it, any potential intervening options to resume operation at other points on the stack between the raising of the condition and the handling of an error. Any such opportunities to resume operation will have lost their chance to exist.

"Well, too bad," these languages would say. "If they wanted a chance, they could have handled the error." But the thing is a lot of the business of signaling and handling conditions is about the fact that you only have partial knowledge. The more uncertain information you are forced to supply, the more your system will make bad decisions. For best results, you want to be able to defer decisions until all information is available. Simple-minded exception systems are great if you know exactly how you want to handle things ahead of time. But if you don't know, then what are you to do? Common Lisp provides much better mechanisms for navigating this uncertain space than other languages do.

So in Common Lisp you can say "I got an argument of the wrong type. Moreover, I know what I would do with an argument of the right type, I just don't happen to have one or know how to make one." Or you can say "Not only do I know what to do if I'm given an argument of the right type (even at runtime), but I even know how to store such a value so they won't hit this error over and over again." In other languages, if the program doesn't know this correctly typed value, even if you (the user) do know it at runtime, you're simply stuck.

In Common Lisp, you can specify the restart mechanism separately from the mechanism of choosing among possible restarts. Having this ability means that an outer part of the program can make the choice, or the choice can fall through to a human user to make. Of course, the human user might get tired of answering, but in such a case, they can wrap the program with *advice* that will save them from the need to answer. This is a much more flexible division of responsibility than other languages offer.

Daydreaming

As programmers, let us daydream for a moment about an ideal system for handling special situations in our code.

Some of us already know the exception handling systems of popular programming languages, such as C++ or Java. When an exception is thrown, it "bubbles up," immediately travelling to the caller of the offending code, or the caller's caller, and so on. Not only is the state of the program destroyed during the process of attempting to find a handler for that exception, but also, if no handler is available, the program will be in no state to continue, since all of its stack will have been unwound (lost). The program will crash with no means of recovering, only in certain scenarios leaving a core dump in its wake.

We could do better than that.

Let us imagine a system where stack unwinding is a choice, not a diktat—a system where, when an exceptional situation is detected, the stack is wound further instead of being immediately unwound. Such a system, at the time of signaling an error, would inspect the program state for all handlers that are applicable to that situation. It would then execute them and let them choose either to repair the program state or to execute some code. Finally, it would transfer control to a known place in the program, and continue execution from there.

Such a system could even use that mechanism for situations which are not traditional errors, but which would nonetheless benefit from being treated in such a way. A program could, at some point during its execution, announce that a given condition has just happened, as well as listing the callbacks suitable for that particular situation.

These callbacks might have been declared completely outside the given piece of code. They might have been passed into it dynamically, also from outside, as applicable to the wider context in which the given code is run. The program could then call all of these callbacks in succession. Or, it could perform additional logic to check if a callback

applies to a given situation, and only then call it. Or, it could attempt to call a particular class of callbacks first, before resorting to a less preferable method of recovery.

Our imagined system would also be closer to perfect if an error situation, once it ran out of means of being handled, would not crash the program. Instead, an internal debugger would pop up and allow the programmer to inspect the full state of the program when the error happened, offering them some predefined means of handling the error. It could also allow them to issue arbitrary commands for inspecting, recovering, or even modifying the program's state.

Ideally, we could express such a system in the self-same programming language for which we design it. Even more ideally, we could construct such a system from scratch in that programming language and seamlessly integrate it into the rest of that language, ensuring that its operation will be able to fit well with any particular domain for which a program is meant to work.

The good thing about such a daydream is that you can experience it even outside the world of your daydreams.

This reality is called the Common Lisp condition system. It is the main focus of this book.

Preface

The Common Lisp condition system has its early roots in PL/I, a programming language designed by IBM in the 1960s. It was promptly adopted in Multics, an operating system from the same time period, as its main programming language. PL/I had an initial condition mechanism whose traits were, among others: separation of condition *signaling* from condition *handling* (in detail: separation of *handler invocation* from *stack unwinding*), the ability to resume an erring computation, the ability to associate a block of data with a signaled condition, and the provision of default handlers for otherwise unhandled conditions. These four ideas ended up inspiring extensions made in the MIT dialect of Lisp Machine Lisp; from there, they have given rise to the contemporary Common Lisp facilities of, respectively, *handlers*, *restarts*, *condition types*, and the *debugger*.

The first book to describe Common Lisp as a dialect was *Common Lisp the Language, First Edition* (also called "CLtL1"), by Guy L. Steele, Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. From there, the standardization work continued, and took two slightly differing branches: one of them came from private work of Guy L. Steele, who described the ongoing standardization work by the X3J13 committee in the book *Common Lisp the Language, Second Edition* (also called "CLtL2"). The other branch, and the final version of the language, was defined in the ANSI document ANSI INCITS 226-1994 and standardized in 1994, giving birth to ANSI Common Lisp as we know it nowadays. (Modern Common Lisp implementations that purport to conform to the ANSI specification also include many CLtL2-only features as language extensions.)

With such a background, the Common Lisp condition system is organized differently than the exception handling mechanisms in other programming languages, such as C++, Java, or Python. Despite being built of ordinary control mechanisms that are also present in other languages, the Common Lisp condition system includes concepts which are glaringly missing from those others. It is as if the exception systems in other languages are in a rush to exit the context of the error immediately and transfer control to a point higher on the call stack, where less contextual information and fewer control options are available.

This difference often creates misunderstanding and confusion about how the CL condition system functions and what are its possible and actual use cases. Knowledge about the internal functionality and utility of all aspects of the condition system is not widely spread. This lack of knowledge, too often, dissuades CL programmers from utilizing the condition system, even in situations where using it would provide clear overall benefit.

The hope of improving this situation is the *raison d'être* of the book whose introduction you are reading right now. *The Common Lisp Condition System* is one more attempt to explain the functionality and utility of Common Lisp conditions, handlers, and restarts, using an approach that is different from former work on the topic. We gratefully acknowledge the many previous authors who have, over the years, used various approaches to explain the Common Lisp condition system to novice and intermediate CL programmers. An incomplete list includes Kent M. Pitman, Peter Seibel, Chaitanya Gupta, Justin Grant, Timmy Jose, Jacek Złydach, and Robin Schroer as well as the collaborative effort behind the Lisp Cookbook; nonetheless, this book takes a fresh and novel approach which we feel is needed and wanted and which will bring a new level of understanding and appreciation to both old-timers and newcomers alike.

Instead of describing the condition system from an outside perspective, we implement the basics of the handler and restart subsystems piecewise from scratch. We take this approach to demonstrate how such subsystems can be bootstrapped from certain key basic language features including dynamic variables and a few simple data structures. We then rewrite these examples to use standard CL functionality to draw parallels between our implementation and the CL-provided tools. In addition, we elaborate on all aspects of the condition system which are defined by the standard, including the lesser-used and not commonly understood ones. At the end, we also propose a few portable extensions to the condition system which augment the tools defined by the ANSI CL standard.

TCLCS is intended to be read in order. Some topics are simplified or given very brief treatment in the earlier chapters, only to be defined, detailed, and expanded upon later in the book. Therefore, it would be unwise to attempt to use this entire book as a reference for the condition system and CL control flow operators. Happily, we have provided such a reference, replete with examples, as an appendix.

In addition, TCLCS is meant to present the basic flow of working with CL as an interactive programming language. We frequently test expressions in the read-eval-print loop (REPL) before defining them as concrete functions. Conversely, we sometimes test concrete functions in the REPL immediately after defining them as well. We also make use of Common Lisp's ability to redefine functions at runtime in order to define function "stubs" that help us test our programs.

The code shown in the first part of this book is published as a Git repository. The code in the second part is adapted from the Portable Condition System repository. The code listings contained in this book are rendered in black-and-white; if the reader prefers syntax highlighting/colorizing, then the code available at the aforementioned links can be viewed and edited in a preferred tool such as Gnu Emacs which performs such highlighting automatically.

TCLCS assumes that you know the basics of the C programming language and some basics of Common Lisp. If you're familiar with neither of these languages, but do know some fundamentals of computer programming, you may still be able to glean the meanings of most of the operations from the text. In case of questions, feel free to throw them at the author via email.

The book is divided into two parts. The first part will construct the basic components of the condition system from scratch, starting from the concept of dynamic variables that makes the condition system possible and then going through the handler and restart subsystems, assertion operators, and other details of a condition system. We will use a storytelling approach in the book to describe the scenarios in which a condition system may prove useful and then write our code.

The second part of the book will use the concepts from the first part to implement an ANSI-compliant condition system from scratch. Reading the first part of the book is not strictly necessary to understand the second; however, the concepts established in the first part should prove useful for someone who has not worked with a condition system before.

Hall of Fame

The Common Lisp Condition System is a work of the Common Lisp community. Without many people who contributed changes, reviews, ideas, and former work, this book would never have come into existence or enjoyed such level of polishing and review.

Thanks to Lonjil, John Cowan, Michael Reis, Aleksandr Treyger, Elias Mårtenson, Adam Piekarczyk, Nisar Ahmad, Robert Strandh, Daniel Kochmański, Michael Fiano, cage, Bike, Nicolas Hafner, Hayley Patton, Grue, Gnuxie, Tom Peoples, Marco Heisig, Georgiy Tugai, Selwyn Simsek, Shubhamkar Ayare, Philipp Marek, Vsevolod Dyomkin, vindarel, François-René Rideau, Jerome Onwunalu, sarna, Travis Sunderland, Jacek

Podkański, Dave Cooper, orbifx, Kent M. Pitman, Peter von Etter, Florian Margaine, and Paul F. Dietz for various remarks, suggestions, reviews, and fixes to the text of this book and to the code of `trivial-custom-debugger` and the Portable Condition System.

Thanks to Kent M. Pitman, Peter Seibel, Paul Graham, David Lamkins, Robin Schroer, Chaitanya Gupta, Justin Grant, Timmy Jose, and Jacek Złydach as well as the Lisp Cookbook contributors for their collective work on evolving, defining, teaching, and writing about the Common Lisp condition system.

Thanks to Vsevolod Dyomkin, Daniel Kochmański, and Georgiy Tugai for providing input about practical examples of using the condition system beyond exception handling.

Thanks to Marco Heisig, Michael Raskin, and Gilbert Baumann for their help with the example implementation of dynamic variables in the C programming language.

Thanks to David A. Moon, Kent M. Pitman, and Bernard S. Greenberg for providing information about the history of the Common Lisp condition system and its predecessors.

Thanks to Alexander Artemenko, Paul M. Rodriguez, ein Strauch, and Chris Bøg for financial support on Patreon and GitHub Sponsors. (If you like what I write, feel free to tip me there yourself.)

Thanks to Nina Palińska for all kinds of personal support during the process of working on the text.

Thanks to you—the reader—for deciding to take a peek at the contents of this book.

CHAPTER 1

Basic concepts

Before diving into the depths of the Common Lisp condition system, we will first introduce three programming concepts that collectively form the foundation of the condition system: *dynamic variables*, *performing non-local transfers of control*, and *lexical closures*. Understanding these lower-level techniques is important to understanding the functioning of the CL condition system as a higher-level mechanism; therefore, readers who are already proficient with these techniques may consider skipping the relevant sections and continue from the next chapter of the book.

Other programming languages tend to implement some of these concepts in various ways. For completeness, however, this book explains each of these concepts from the very beginning, since utilizing all three of them in combination to form a complete condition system is unique to CL as a language.

1.1 Dynamic variables

We will begin with the very feature of CL that makes the handler and restart subsystems possible and in fact easy to implement: dynamic variables. Instead of defining the term "dynamic variable" directly, we will try to convey its meaning through examples instead and only provide the definitions afterward.

1.1.1 Dynamic variables in C

Let's start with an example in C—the *lingua franca* of our profession, in a way.

```
int x = 5;

int foo() {
  return x;
}

foo(); // -> 5
```

In the preceding example, we have a global variable named `x`, with the function `foo` returning the value of that variable. The result of calling `foo()` will be the integer `5`.

```
int x = 5;

int bar() {
  return x;
}

int foo() {
  return bar();
}

foo(); // -> 5
```

In the preceding example, we have a global variable named `x`, with the function `bar` returning the value of that variable and the function `foo` calling the function `bar`. Compared to the previous example, we have added a level of indirection. But the result is still the same: calling `foo()` still gives us `5`.

```
int x = 5;

int bar() {
  return x;
}

int foo() {
  int x = 42;
  return bar();
}

foo(); // -> 5
bar(); // -> 5
```

The preceding example adds one more change: a new variable binding for `x` is introduced in the body of `foo`, with a value of `42`. That variable is *lexically scoped*, though, meaning that it is only effective within the body of the block in which it is declared. The variable `x` is not used within the block of function `foo`; the definition `int x = 42;` is effectively unused.

```
dynamic_var(int, dynamic_x, 5);

int bar() {
  return dynamic_x;
}

int foo() {
  dynamic_bind(int, dynamic_x, 42) {
    return bar();
  }
}

bar(); // -> 5
foo(); // -> 42
bar(); // -> 5
```

The preceding example adds one more twist: our global variable is now defined via `dynamic_var`. This means that the scope of this variable is no longer lexical, but *dynamic*; the scope of the variable is not affected by curly braces (or a lack thereof), but by the runtime *environment* in which that variable is accessed.

This modification is enough to cause `foo()` to return `42`, even though calling `bar()` alone still returns 5!

(The operators `dynamic_var` and `dynamic_bind` are not a part of the C standard; in fact, C has no intrinsic notion at all of dynamic variables. We describe a means of adding these operators to the C language in an appendix to this book.)

At runtime, each new binding of a dynamic variable creates a new *environment*, in which the name of that variable (in this case, `dynamic_x`) is bound to a value (in this case, `5` for the global binding and `42` for the binding in `foo()`). These environments are always accessed in order, from the last defined one to the first defined one; looking up the value of a dynamic variable consists of going through the environments until we find the first environment (i.e., the most recent one chronologically) that has the binding for that dynamic variable. The value of that binding is then accessed.

(One can notice that `x`, in this example, has been renamed to `dynamic_x`. Such a distinction in naming is important. Dynamic variables have different semantics from standard variables and should therefore be visually distinguished from normal, lexically scoped variables.)

If one knows the stack data structure, then one can think of the set of environments as a stack. Variable bindings are pushed onto this stack such that the most recent one is on top. If the system later has need to search that stack for a given variable binding, then it looks from top to bottom (i.e., starting with the most recent) for the first instance of that variable binding. Therefore, in a program without any dynamic variables, the stack of environments is empty:

```
            (nothing here)

-------------b-o-t-t-o-m-------------
```

(It therefore would result in an error to attempt to access a dynamic variable named `dynamic_x`; it is unbound, meaning there is no value whatsoever associated with it!)

But, if we wanted to illustrate the top-level environment from the earlier example with `dynamic_x`, it would look like this:

```
        --------------------
        |   dynamic_x: 5   |
-------------b-o-t-t-o-m-------------
```

There is only one (global) dynamic environment that contains the binding of the variable `dynamic_x`. If we called the function `bar()`, which returns the dynamic value of `dynamic_x`, then the system would access that environment, and it would return `5`.

The situation changes, however, if we call `foo()`, because of the `dynamic_bind(int, dynamic_x, 42)` binding found there. `dynamic_x` has been declared dynamic at the top level, which "infects" all future binding of that variable; it means that it becomes a dynamic variable, and its value therefore becomes stored on the environment stack when `foo()` is called. The situation inside `foo()`, after *rebinding* the dynamic variable but before calling `bar()`, looks like this:

```
        --------------------
        |   dynamic_x: 42  |
        --------------------
        |   dynamic_x: 5   |
-------------b-o-t-t-o-m-------------
```

When `bar()` gets called, it returns the value of `dynamic_x`. The system looks it up in the environment stack, starting from the top. The first environment that contains a binding for `dynamic_x` is the one with the value `42`, which `bar()` then returns to `foo()` and which `foo()` then returns to the original caller.

This stack-like lookup mechanism means that, with this simple change in declaration, calling `bar()` still gives us `5`, but calling `foo()` gives us `42` instead of `5`!

It also means, indirectly, that dynamic variables give us a way to affect the environment of functions that we execute *dynamically*. "*Dynamically*," in this sense, means "depending on *where* a given function was run from." If we run it from the top level or from a `main()` function that itself has no dynamic bindings, then only the global dynamic bindings shall be in effect; if it gets called from another scope in the code, then it will be called with whatever dynamic bindings this particular scope may have defined *on top of* all the dynamic bindings that have been defined in previous dynamic environments.

The difference between lexical variables and dynamic variables can be summarized in one more way: *a lexical variable cannot be seen outside its textual scope*, while *a dynamic variable cannot be seen outside its runtime scope*. When a dynamic scope ends, the *environment* defined in it is discarded: after runtime control leaves `foo()`, the dynamic environment defined in `foo()` is removed from the environment stack, leaving us in a situation similar to the previous one:

```
        --------------------
        |   dynamic_x: 5   |
-------------b-o-t-t-o-m-------------
```

This dynamic scoping means that if we were to try to call `bar()` again immediately after calling `foo()`, then the dynamic environment created inside `foo()` would not affect the subsequent execution of `bar()`. When `foo()` finishes, it cleans up the dynamic environment that it created, and therefore `bar()` sees, and can access, only the global environment which is remaining.

This operating principle means that the dynamic environment is preserved even in situations with more deeply nested function calls. If we have a function `foo` that calls `bar` that calls `baz` that calls `quux`, and we define a `dynamic_var(int, dynamic_x, 42)` at the top of `foo()`'s body, and then we try to access `dynamic_x` inside `quux`, then the access is going to work. Of course, `bar` and `baz` can introduce their own `dynamic_binds` of `dynamic_x` that are then going to affect the value that `quux` will find; all that is necessary for a dynamic variable access to work *always* is at least one binding for that particular variable, which—in our example implementation in C—is guaranteed to exist.

Nothing prevents us from using multiple dynamic variables either. For simplicity, we will use an environment model in which a single environment frame always holds only one variable-value binding. Let us assume that we have three variables, `dynamic_x` (which is later rebound), `dynamic_y`, and `dynamic_z`. At one point in execution of the program, the environment stack is going to look like the following:

```
        ------------------------------
        | dynamic_z: [2.0, 1.0, 3.0] |
        ------------------------------
        |      dynamic_x: 1238765    |
        ------------------------------
        |      dynamic_y: "Hello"    |
        ------------------------------
        |        dynamic_x: 42       |
------------------b-o-t-t-o-m--------------------
```

Accessing a `dynamic char* dynamic_y` will return `"Hello"`, accessing a `dynamic int dynamic_x` will return `1238765` (it was rebound once, and the previous value of `42` is ignored), and accessing a `dynamic float dynamic_z[3]` will return the vector `[2.0, 1.0, 3.0]`. Note that I mentioned "accessing," which means not just getting the values but *setting* them too. It is possible, for example, to execute the following body of code that sets the values of these variables:

```
{
  dynamic_x = 2000;
  dynamic_y = "World";
  dynamic_z[1] = 123.456;
}
```

The resulting environment is going to look like this:

```
     ----------------------------------
     | dynamic_z: [2.0, 123.456, 3.0] |
     ----------------------------------
     |        dynamic_x: 2000         |
     ----------------------------------
     |       dynamic_y: "World"       |
     ----------------------------------
     |         dynamic_x: 42          |
----------------b-o-t-t-o-m--------------------
```

It is noteworthy that the last environment, containing the original binding for `dynamic_x`, was not affected by the operation of setting. When the system tried to set the value of `dynamic_x`, it searched for the first environment where `dynamic_x` was bound from top to bottom and modified that environment only.

One consequence of this principle is that only the most recent binding for each dynamic variable is visible to running code; running code cannot access any environments or bindings defined previously. This approach is useful in case we expect that some code might modify the value of a dynamic variable; for instance, if we want to protect the current value of `dynamic_x` from modification, then we can define a new binding in the form of `dynamic_bind(int, dynamic_x, dynamic_x)`—a binding which is then going to be affected by any subsequent `dynamic_x = ...` setters.

To sum up, there are three parts of syntax related to dynamic variables:

- Defining a variable to be dynamic
- Creating new dynamic bindings
- Accessing (reading and/or writing) the most recent dynamic binding

The only sad thing about dynamic variables in C is that the preceding code that uses `dynamic_var` and `dynamic_bind` variables will not compile by default. As mentioned earlier, that code is invalid, due to C having no notion of dynamic variables whatsoever. Therefore, even though these code examples (hopefully!) illustrate the concept of dynamic variables, they will not work out of the box with any known C compiler on planet Earth. However, such notion can be added to C by means of programming the C *preprocessor*, using the technique that we will describe now.

1.1.1.1 Implementing dynamic variables in C

All fibbing aside though, it is in fact possible to implement dynamic variables in C and other languages that do not have them in their standard. The technique involves the following steps: save the original value of a variable in a temporary stack-allocated variable, set the variable with the new value, execute user code, and then restore the variable with its original value.

This technique is sometimes used in practice to implement dynamic variables (e.g., in Emacs C code), and it replaces the explicit, separate stack of environments with an *implicit* stack, embedded within the temporary variables that are allocated by the program.

Example:

```
int x = 5;

int bar() {
  return x;
}

int foo() {
  int temp_x = x;  // Save the original value of X
  x = 42;          // Set the new value to X
  int ret = bar(); // Execute user code and save its return value
  x = temp_x;      // Restore the original value of X
  return ret;      // Return the user code's return value
}

foo(); // -> 42
bar(); // -> 5
```

We are binding the dynamic variable by means of creating new lexical variables and temporarily storing the old dynamic values there. The original lexical variable gets overwritten with a new value and then restored when control is leaving that runtime scope.

This approach is messy and prone to errors due to the amount of assignments required; it is, however, possible to hide them by writing macros for the C preprocessor. The reader can consult an appendix to this book for an example implementation of `dynamic_var` and `dynamic_bind` that uses a variant of the preceding technique.

1.1.2 Dynamic variables in Common Lisp

In contrast to C, certain languages, notably Common Lisp, do have built-in support for dynamic variables. We will be restricting our discussion to Common Lisp (CL) for the rest of the book. CL is an ANSI-standardized dialect of the Lisp programming language that supports various programming paradigms, including procedural, functional, object-oriented, and declarative paradigms. CL has dynamic scoping as part of its ANSI standard, and it should be possible to execute all of the examples here in any standard-conforming CL implementation. This happy fact frees us from the need to depend on any particular CL implementation or compiler.

Let us begin with a simple example showing how Lisp variables work in general. Contrary to C, it is possible in Lisp to define a function that has access to lexical variables which themselves are defined outside that function definition, substantially reducing the need for global variables in a program (at least non-dynamic ones).

```
(let ((x 5))
  (defun foo ()
    x))

(foo) ; -> 5
```

In the preceding example, we have a lexical variable named `x`, with the function `foo` returning the value of that variable. Exactly as in the related C example, the result of calling `foo` will be `5`.

```
(let ((x 5))
  (defun bar ()
    x)
  (defun foo ()
    (bar)))

(foo) ; -> 5
```

After adding a level of indirection to accessing the variable, this example still behaves the same way as in C.

```
(let ((x 5))
  (defun bar ()
    x)
  (defun foo ()
    (let ((x 42))
      (bar)))
  (defun quux ()
    (let ((x 42))
      x)))

(foo) ; -> 5
(quux) ; -> 42
```

If we experiment with introducing new lexically scoped variables in Lisp, the result will still be consistent with what C produces: calling `(foo)` gives us `5` (the variable definition `(let ((x 42)) ...)` inside the body of `foo` going effectively unused), and calling `(quux)` gives us the shadowing value `42`.

```
(defvar *x* 5)

(defun bar ()
  *x*)

(defun foo ()
  (let ((*x* 42))
    (bar)))

(foo) ; -> 42
(bar) ; -> 5
```

The preceding example changes two things. First, the name of our variable is now modified from `x` to `*x*`—the "earmuff notation" which in CL is the conventional notation for dynamic variables. Second, the variable we define is now global, as specified via `defvar`.

(It is possible to neglect the earmuff notation and do things like `(defvar x 5)`, since the earmuffs themselves are just a part of the symbol's name and their use is not mandated by any official standard. Earmuffs were introduced as a convention for separating global dynamic variables clearly from other symbols, since unintentionally rebinding a dynamic variable might—and most often will—affect code that executes deeper in the stack in unexpected, undesired ways.)

The environment stack works in the same way as in the C example: calling `(bar)` gives us the original, top-level value `5`, but calling `(foo)` will give us the rebound value `42`.

In Common Lisp, it is additionally possible to refer to a dynamic variable *locally*, instead of using a variable that is globally special. If we were to do that, then the following would be an incorrect way of doing that:

```
;;; (defvar *y* 5) ; commented out, not evaluated

(defun bar ()
  *y*) ; this will not work

(defun foo ()
  (let ((*y* 42))
    (declare (special *y*))
    (bar)))
```

Inside `foo`, we locally declare the variable `*y*` to be *special*, which is Common Lisp's notation for declaring that `*y*` denotes a dynamic variable in this context. However, we have not done the same in the body of `bar`.

Because of that omission, the compiler is free to treat *y* inside the body of bar as an undefined variable and to produce a warning:

```
; in: DEFUN BAR
;     (BLOCK BAR *Y*)
;
; caught WARNING:
;   undefined variable: COMMON-LISP-USER::*Y*
```

(Note that we no longer use the symbol *x* as the variable name, since the previous example proclaimed it to name a global dynamic variable. If we wanted to "undo" that fact and be able to proclaim *x* as dynamic locally, we would need to (unintern '*x*) in order to remove the symbol *x*, along with that global proclamation, from the package in which we are currently operating. Removing the symbol will not affect the Lisp system negatively; it will be re-created the next time we use it, as soon as the reader subsystem reads it.)

The correct version is:

```
;;; (defvar *y* 5) ; commented out, not evaluated

(defun bar ()
  (declare (special *y*))
  *y*)

(defun foo ()
  (let ((*y* 42))
    (declare (special *y*))
    (bar)))

(foo) ; -> 42
```

Similarly, calling (bar) directly (e.g., from the top level) is an error, since the variable *y* does not have a global dynamic binding.

An environment stack with multiple dynamic variables (named *x*, *y*, and *z* in Lisp) may look like the following:

```
        -----------------------------
        |     *z*: #(2.0 1.0 3.0)    |
        -----------------------------
        |       *x*: 1238765         |
        -----------------------------
        |       *y*: "Hello"         |
        -----------------------------
        |         *x*: 42            |
-----------------b-o-t-t-o-m-------------------
```