APPENDIX E

Discussing the Common Lisp condition system

Foreword

Here are several sections of content related to the topic of the Common Lisp condition system that were created once the book already went to the proofing stage. This content was created after the first Hacker News discussion related to the book *The Common Lisp Condition System*. Since it was already too late to add these sections to the body of the book, it was suggested to gather these written words and create a free, online-only appendix for the book that will be available for download from the official Apress website for the book; it could then be linked from the end of Chapter 4, where these sections were originally supposed to go.

To be precise, the contents of this appendix are meant to be read immediately after Section 4.6.4 of the original book. In this appendix, we supplement the closing part of the book with new details and information. We explain several aspects of working with the condition system in practice that are not immediately obvious; we describe practical use cases where a condition system serves well, compare the benefits of a condition system with design approaches popular in other programming languages, and describe how the condition system augments other aspects of a CL system in beneficial ways. Finally, we elaborate on situations where having a living condition system embedded in a running program really shines, and we show the most important distinctive benefits that a condition system can make in such cases.

Thanks to `eismcc`, `lonjil`, `pierrebai`, `kazinator`, `db48x`, `lucozade`, `pfdietz`, `guenthert`, `TeMPOraL`, `wtetzner`, `throwaway_pdp09` (anonymous HN poster), `akamaozu`, `zokier`, `Archit3ch`, `Akronymus`, `divs1210` (Divyansh Prakash), `aidenn0`, `mark_l_watson`, `foobar_`, `mst` (Matt S. Trout), `nahuel0x`, `graememcc`, `jshaqaw`, `fovc`, `minerjoe`, `gumby`, `drcode`, `jwr`, `lukashrb`, `kqr`, `hardeeparty`, `zeveb`, `p_l`, `plafi`, dmux, tester34, protomyth, rags2riches, dan-robertson, mjbishop, steveklabnik, MycroftJones, dragonwriter, and daly for participating in the aforementioned Hacker News discussion, providing valuable insight, asking tough questions, and therefore providing insight and content for the book.

Chapter 4: Continued

Section 4.6: Continued

4.6.5 Debugger-oriented programming

One could argue that programming by means of purposefully invoking errors and then immediately fixing them is unusual indeed. It is, however, particularly common and effective in Smalltalk, which is another image-based programming language that includes a full debugger along with its complete development environment—and this programming style is also possible in Lisp.

The idea of this programming technique is to write an invalid code snippet and attempt to execute it, causing an immediate entry into the debugger for one of multiple reasons: unbound variables, undefined functions, type errors, argument count mismatches, and so on. The programmer can then use the debugging facilities of the programming language to "fix" the error and continue program execution. This process can be repeated multiple times until the programmer is satisfied with how the code works.

A reader familiar with the concept of test-driven development (TDD) may notice a similarity to the "red-green-refactor" paradigm used as one of its foundations: the command passed to the REPL can be seen as a failing unit test, which is the "red" phase; the process of working in the debugger in order to continue execution can be seen as fixing the code to pass this unit test, which is the "green" phase; finally, the moment of adjusting the written code better to suit the programming style is the literal definition of the verb "refactor."

We can show an example of such programming flow. Let us attempt to define a small Lisp package, greeter, which will export simple functionality for greeting people. We will start at a fresh Lisp REPL in our hypothetical Lisp implementation. In that REPL, we will immediately attempt to refer to the greeter package, and obviously, the Lisp system is going to alert us to the fact that the package does not yet exist:

```
CL-USER> (greeter:greet "Thomas")
;;
;; Debugger entered on READER-PACKAGE-ERROR:
;; The name "GREETER" does not designate any package.
;;
;; Available restarts:
;; 0 [CONTINUE ] Use the current package, COMMON-LISP-USER.
;; 1 [RETRY    ] Retry finding the package.
;; 2 [USE-VALUE] Specify a different package
;; 3 [UNINTERN ] Read the symbol as uninterned.
;; 4 [SYMBOL   ] Specify a symbol to return.
;; 5 [ABORT    ] Return to the top level.
;;
;; Enter a restart number to be invoked
;; or an expression to be evaluated.

[1] Debug>
```

As seen earlier, several restarts make themselves available for continuing execution. If we inspect the backtrace, we can ascertain that the error was signaled while reading the Lisp form passed to the REPL; interning the greeter:greet symbol failed due to the greeter package being unavailable. The continue restart will use the current package, common-lisp-user, which is not what we want; for similar reasons, we do not want unintern. We should be able to use the retry, use-value, or symbol restarts, but only once the greeter package exists. Therefore, let's create it:

```
[1] Debug> (make-package :greeter :use '(#:cl))
#<PACKAGE "GREETER">

[1] Debug> (invoke-restart 'retry)
;;
;; Debugger entered on READER-PACKAGE-ERROR:
;; Symbol "GREET" not found in the GREETER package.
;;
;; Available restarts:
;; 0 [CONTINUE] Use symbol anyway.
;; 1 [RETRY   ] Retry looking for the symbol.
;; 2 [ABORT   ] Return to top level.
;;
```

```
;; Enter a restart number to be invoked
;; or an expression to be evaluated.
```

Again, we land in the debugger. This time, the package has been found, but it does not contain a symbol named `greet`. We can manually intern that symbol into the `greeter` package to fix this error and then non-forcibly retry looking for the symbol via the `retry` restart:

```
[1] Debug> (intern "GREET" (find-package :greeter))
GREETER::GREET
NIL

[1] Debug> (invoke-restart 'retry)
;;
;; Debugger entered on READER-PACKAGE-ERROR:
;; The symbol "GREET" is not external in the GREETER package.
;;
;; Available restarts:
;; 0 [CONTINUE] Use symbol anyway.
;; 1 [RETRY   ] Retry accessing the symbol.
;; 2 [ABORT   ] Return to top level.
;;
;; Enter a restart number to be invoked
;; or an expression to be evaluated.
[1] Debug>
```

Once again, we land in the debugger—the symbol exists in the package, but it is not exported from it:

```
[1] Debug> (export 'greeter::greet (find-package :greeter))
T

[1] Debug> (invoke-restart 'retry)
;;
;; Debugger entered on UNDEFINED-FUNCTION:
;; The function GREETER:GREET is undefined.
;;
;; Available restarts:
;; 0 [CONTINUE      ] Retry calling GREETER:GREET.
;; 1 [USE-VALUE     ] Call specified function.
;; 2 [RETURN-VALUE  ] Return specified values.
;; 3 [RETURN-NOTHING] Return zero values.
;; 4 [RETRY          ] Retry evaluating the form.
;; 5 [ABORT          ] Return to top level.
;;
;; Enter a restart number to be invoked
;; or an expression to be evaluated.
[1] Debug>
```

We land in the debugger *yet again*. (Do not worry—we will continue the fun a bit longer.) This time, accessing the symbol has succeeded, but there is no function named by that symbol. That is expected, since the symbol has just been created and we have not yet defined any functions associated with it.

We will want to use the `continue` restart in order to call the function once it is defined; because of this, the restarts `use-value`, `return-value`, and `return-nothing` are not useful in this particular context. Let us switch to the `greeter` package for convenience (so we do not have to prefix all of its internal symbols with `greeter::`) and define the requisite function, implementing its body—all without leaving the debugger:

```
[1] Debug> (in-package #:greeter)
#<PACKAGE "GREETER">

[1] Debug> (defun greet (name)
             (incf (gethash name *greeted-people* 0))
             (format ";; Hello, ~A! This is the ~:R time I greet you.~%"
                     name (gethash name *greeted-people*)))
GREET

[1] Debug> (invoke-restart 'continue)
;;
;; Debugger entered on UNBOUND-VARIABLE:
;; The variable GREETER::*GREETED-PEOPLE* is unbound.
;;
;; Available restarts:
;; 0 [CONTINUE   ] Retry using GREETER::*GREETED-PEOPLE*.
;; 1 [USE-VALUE  ] Use specified value.
;; 2 [STORE-VALUE] Set specified value and use it.
;; 3 [RETRY      ] Retry evaluating the form.
;; 4 [ABORT      ] Return to top level.
;;
;; Enter a restart number to be invoked
;; or an expression to be evaluated.
[1] Debug>
```

We are progressing. The variable *greeted-people* is indeed unbound, but if Common Lisp has signaled this condition, we can assume that the body of function greet—which uses this variable—has been entered, and we can verify that assumption by inspecting the backtrace. However, for now, let us fix this immediate error by defining the needed variable. We will want its value to be a hash table suitable for utilizing strings as keys:

```
[1] Debug> (defvar *greeted-people* (make-hash-table :test #'equal))
*GREETED-PEOPLE*

[1] Debug> (invoke-restart 'continue)
;;
;; Debugger entered on TYPE-ERROR:
;; The string ";; Hello, ~A! This is the ~:R time I greet you.~%" does not
;; have a fill pointer.
;;
;; Available restarts:
;; 0 [RETRY ] Retry evaluating the form.
;; 1 [ABORT ] Return to top level.
;;
;; Enter a restart number to be invoked
;; or an expression to be evaluated.
[1] Debug>
```

Uh oh. We have made a mistake in the body of greet—we forgot to pass the destination argument to it. Let us promptly fix it by redefining greet and invoking the retry restart to reevaluate the (greeter:greet "Thomas") form:

```
[1] Debug> (defun greet (name)
             (incf (gethash name *greeted-people* 0))
             (format t ";; Hello, ~A! This is the ~:R time I greet you.~%"
                       name (gethash name *greeted-people*)))
GREET

[1] Debug> (invoke-restart 'retry)
;; Hello, Thomas! This is the second time I greet you.
NIL

GREETER>
```

Finally, we have successfully left the debugger, though one issue persists: the formatted output states that this is the *second* time that Tom has been greeted. This is because the first, failed invocation of `greet` managed to increase the greet count even though the `format` call did not succeed.

Let us fix that by cleaning the hash table and quickly testing the function in the REPL:

```
GREETER> (clrhash *greeted-people*)
#<HASH-TABLE :TEST EQUAL :COUNT 0 {10031E5393}>

GREETER>  (greeter:greet "Thomas")
;; Hello, Thomas! This is the first time I greet you.
NIL

GREETER>  (greeter:greet "Thomas")
;; Hello, Thomas! This is the second time I greet you.
NIL

GREETER>  (greeter:greet "Thomas")
;; Hello, Thomas! This is the third time I greet you.
NIL
```

Our code works properly! Now it is time to trace the steps we have made in the REPL and clean up the code we passed for evaluation. We may want, for example, to merge the three calls to `make-package`, `intern`, and `export` into a single `defpackage` form and to modify the `greet` function to increase the greet count only once the `format` call has succeeded. With those tasks done, we are ready to start another development cycle by entering the next desired Lisp form into the REPL, hacking away in the debugger until it actually *does* work in the way we intend, and then cleaning up the hard-earned code we've passed to the REPL and finally saving it to a source file in persistent storage.

This style of programming is useful both during development and while fixing real bugs; a zealous reader might even endeavor to perform purposeful debugger-oriented programming as training for fixing an actual fault in live Lisp software. This opportunity exists because when a Lisp program enters the debugger, the Lisp system is designed in a way that makes it possible to inspect the call stack to find the bug, fix it, and then complete the function call from the middle of a given function at the exact point where the exception happened. The fact that Lisp is an image-based programming language makes it possible to reset variables, redefine functions or class definitions, and create completely new Lisp data on the fly—all the while neither leaving the debugger nor destroying the program state that led to entering the debugger in the first place.

4.6.6 Reading Common Lisp code: Eclector

One of the more commonly encountered issues when attempting to read CL code with the Lisp reader is the fact that some Lisp forms have compile-time side effects that are required for reading further. This is a major issue, for example, when attempting to perform static analysis of CL code or performing mutational fuzz testing. The most prominent examples of this situation are toplevel `defpackage` and `in-package` forms: simply reading these forms is not enough, since their presence in a file directly affects the way in which subsequent forms (and the symbols they include) should be read. Therefore, we would like a graceful way to handle the situation where we read `foo:bar` but no package named `foo` is defined.

One solution to this issue is to program the CL reader. Let us say that, when we are reading CL forms, we would like to represent unknown symbols by a custom structure that we define. Such a structure may contain information present in the textual representation of a symbol: package name, symbol name, and the number of colons that determine how the symbol was accessed. An example implementation of such a structure might look like this:

```
(defstruct unknown-symbol package name colons)

(defmethod print-object ((object unknown-symbol) stream)
  (print-unreadable-object (object stream :type t)
    (princ (unknown-symbol-package object) stream)
    (dotimes (i (unknown-symbol-colons object)) (write-char #\: stream))
    (princ (unknown-symbol-name object) stream)))

CL-USER> (make-unknown-symbol :package "FOO" :name "BAR" :colons 1)
#<UNKNOWN-SYMBOL FOO:BAR>
```

As we noticed in the previous chapter, the Lisp reader often offers multiple system-provided restarts upon encountering symbols that it is unable to read. However, for consistency between implementations and for the ease of introducing our own, we shall use the Eclector library, which implements a portable and highly programmable Lisp reader. In particular, Eclector uses a technique called *clients* in which the default behavior of the application can be fully customized by using an additional argument, the *client*, which is an object of a particular class, and defining methods that specialize on that class. For instance, we may program the way in which Eclector interprets symbols by defining our custom client class named `my-client` and then writing a method on the `eclector-reader:interpret-symbol` generic function that specializes on this client class:

```
(defclass my-client () ())

(defmethod eclector.reader:interpret-symbol
    ((client my-client) input-stream package-indicator symbol-name internp)
  (restart-case (call-next-method)
    (eclector.reader:recover ()
      :report "Return an unknown symbol structure."
      (make-unknown-symbol :package (string package-indicator)
                           :name symbol-name
                           :colons (if internp 2 1)))))
```

The standard behavior for Eclector when a symbol's package does not exist is the same as the standard CL behavior in that case: to signal an error. However, the preceding method adds an additional restart named `eclector-reader:recover`, which instead returns an instance of our custom `unknown-symbol` structure. We may utilize this restart by binding the default client object used by Eclector and introducing a condition handler that invokes this restart directly:

```
(defun test (string)
  (let ((eclector.reader:*client* (make-instance 'my-client)))
    (handler-bind ((eclector.reader:package-does-not-exist
                     #'eclector.reader:recover))
      (values (eclector.reader:read-from-string string)))))
```

Now that we have customized our Lisp reader by utilizing a combination of a custom method, restart, and condition handler, we may test it in order to ensure that it behaves as expected:

```
CL-USER> (test "foo:bar")
#<UNKNOWN-SYMBOL FOO:BAR>

CL-USER> (test "foo::bar")
#<UNKNOWN-SYMBOL FOO::BAR>

C>-USER> (test "(defun foo:bar (baz:quux fred:frob) (+ baz:quux fred:frob))")
(DEFUN #<UNKNOWN-SYMBOL FOO:BAR> (#<UNKNOWN-SYMBOL BAZ:QUUX> #<UNKNOWN-SYMBOL FRED:FROB>)
  (+ #<UNKNOWN-SYMBOL BAZ:QUUX> #<UNKNOWN-SYMBOL FRED:FROB>))
```

The fact that Eclector is programmable in such a manner also allows for handling other situations. One can imagine that such programmability may be used as a part of a load-on-demand system, in which, upon reading `foo:bar`, the handler may look up `foo` in some list of packages provided by various Lisp libraries and automatically load the correct library, and then attempt to read the symbol again now that the package is (hopefully) defined.

Note that the reader itself does not have to know about these various ways of dealing with reader errors; it simply needs to provide the restarts, so that various handlers can instruct it on how to proceed to achieve the particular goal defined by the programmer.

4.6.7 Preserving errors, fresh: debuggers and networked debuggers

When one of the threads of a multithreaded server application encounters an unhandled error, then the typical "industry standard" course of action is: log all available information about the request and program state to some sort of log file or crash dump, return a HTTP 500, and start waiting for another request. However, another idea of handling errors in a multithreaded environment is based on the fact that multiple threads waiting on input/output are relatively cheap to host nowadays.

The technique allows the worker thread that signaled the error to block and wait; a new worker thread is then spawned to perform work in its place. Such a thread may wait for the programmer to connect to the server (e.g., by a remote Swank connection) and present the debugger window, allowing them to introspect the state of the thread and resolve the situation. Additionally, the worker thread may "finish" its work, for example, by informing the client of the internal server error, before going to sleep and waiting for the programmer to resolve the situation.

A more exotic variant of the preceding situation involves allowing the thread that has entered the debugger to connect to a command-and-control server and wait for commands; this facilitates fully remote debugging of worker machines by issuing commands to the C&C server. These commands are then routed to the faulting thread on the proper machine. Such interconnection may be achieved by using software such as `swank-client` or `swank-crew`.

An *even* more exotic variant of the aforementioned implements the debugger in a web interface of the application. As we have demonstrated earlier, a complete CL debugger can be implemented in user code with a single portability library; therefore, nothing prevents us from implementing that debugger as code that sends a series of

HTTP or WebSocket messages to a web browser that uses client-side scripting to spawn a graphical Lisp debugger; such approach might be useful for web applications that are meant to be general front ends for Lisp images, such as some McCLIM applications.

Since this approach inherently involves networking, the user needs to ensure that malicious actors cannot exploit it. Swank connections should be hidden behind encrypted SSH tunnels; HTML-based debuggers and REPLs should only be available to website administrators; finally, the Lisp application should impose a limit on the number of threads allowed to go into "sleep mode," since otherwise a bug in the application may be exploited to form a denial-of-service attack by spawning a massive number of threads, each waiting to be interactively debugged. It's also possible to use a global debug variable to control the level of debugging, for example, to control whether the interactive debugger gets invoked or the system simply logs the error, informs the client of the error, and moves on.

4.6.8 Preserving errors, pickled: saving Lisp images

It is also possible to utilize the condition system in the context of another functionality provided by all CL implementations: saving custom Lisp images. Let us imagine a long-running Lisp image with some sort of server running on it (e.g., a HTTP application with WebSocket sessions). The developers of that application may decide that they want to maximize their debugging ability by preserving as much of the erroneous state as possible while keeping the server running so as to not break other users' sessions, which would cause reconnections.

This can be achieved by programming the toplevel error handler in a way that forks the Lisp process (using the system `fork()` routine); this will effectively spawn another Lisp process, independent of the parent. The parent process can, for example, send an HTTP 500 error or communicate an error via the proper WebSocket connection, and meanwhile the child process can go ahead and dump its own state into a core file, allowing the programmers to rerun that Lisp image at a later time for debugging. This provides the advantage of being able to use the internal CL debugger and inspector to analyze the state of the Lisp image as it was at the moment of error.

The main issue when working with this technique is the fact that `fork()` is not well defined in multithreaded code; only the forking thread will be preserved, meaning that other threads may have been stopped inside their critical sections with no `unwind-protect` having executed, which might cause issues when debugging. Another issue is preserving the state of the program stack, as it is destroyed in the process of dumping the Lisp image, and it cannot be fully restored when the image is thawed. However, portability libraries such as Dissect allow for saving the stack state as normal Lisp data that can then be inspected using the system inspector.

4.6.9 Preserving errors, distilled: automatically creating unit tests

Yet another way to preserve error cases lends itself particularly well to servers which are written in a functional style and therefore manage all of their state explicitly. It utilizes the fact that CL is a homoiconic language, in which it is easy for the system to write Lisp code to a file in a way where that code can then be read back in order to produce given results; in other words, homoiconicity is the “code-is-data” approach that Lisp is sometimes praised for.

If the server state at the moment of error is fully explicit (or if the programmer has the patience to make it so), then the programmer will be able to know the exact input that caused the server to crash. This means that it is easily possible to generate a Lisp form that can act as a unit test to reproduce that particular failure. This unit test can then be refactored properly by the programmers and included in the regression test suite for the server to ensure that this particular fault has been fixed and stays fixed in the future.

4.7 Condition system as a design choice

The main difference between the Common Lisp condition system and the exception handling systems of other programming languages, and the most impressive capability of the Common Lisp condition system, is its ability to preserve the stack by default.

Traditional exception systems unwind the stack all the way to the first matching “catch” block, which has the full responsibility of handling the exception; at the point of catching the exception, the stack has already been unwound, which means that there is no possibility to continue execution from the point where the exception was thrown. If anything, by the time the exception is caught, the only possibility is to terminate a given calculation politely and/or restart it from scratch.

In CL, one can implement multiple condition handlers that *decline* to handle the exception (by returning normally instead of performing non-local exits) and instead execute some additional code that may, for example, prepare the system for the fact that a condition may be handled; a condition handler far up the stack may then decide which restart procedures—if any—it wants to invoke. From such a perspective, traditional exception systems may seem like very crippled versions of the Common Lisp condition systems.

In these other languages, when an error happens, the stack is automatically unwound and therefore destroyed; in Lisp, when an error happens, the stack is automatically wound further, and the code executed in this way has a full choice of whether or not to unwind the stack, and where exactly to unwind it to. This information is provided in the dynamic environment in which a given piece of code is running, and this environment is preserved by way of the preserved stack.

It is also noteworthy that this aspect of the condition system is fully independent from Lisp's homoiconicity; rather, it is a consequence of the way other programming languages are designed. For instance, when one divides by zero in a Java program, then there is *nothing* carved in stone which would prevent the language from winding the stack further and executing some code that will analyze the dynamic environment in which the error happened, calling all error handlers found in the dynamic environment, and then—if no handler transferred control outside the error site—either entering an interactive debugger of some sort or giving up and crashing. However, Java goes a different way: it immediately destroys the stack by throwing an exception. That is a design choice which was made by the Java creators; Lisp's homoiconicity has nothing to do with this fact, as can be demonstrated by the multiple independent implementations of condition systems in non-homoiconic languages that we have mentioned earlier.

4.8 Internal debugger as a design choice

In addition to considering a condition system as one design choice of a language, we may also consider a somewhat orthogonal choice: that of whether to include a debugger in a running program or to have an external debugger that is "attachable" to said program.

Let a batch language be a language wherein a program is compiled from source files by a compiler external to said program itself and which produces executable binary files. Many batch languages, such as C, C++, Rust, and Haskell, explicitly provide little or no runtime information about data or functions used in the program at runtime. On one hand, this allows for various optimizations that would not be possible if runtime information about objects should need to be preserved; on the other hand, it means that debugging a program is only possible in the presence of an external debugger that analyzes the live program (or its "dead" crash dump) from outside, optionally using information about function names and data structures passed from outside the program (or sometimes from an additional debug section stored inside the program's binary).

Of utmost importance regarding the topic of exception handling is the limitation that designing the language in such a way makes it impossible to attach a debugger. Even if one introduced a debugger routine that, just like the Lisp debugger, halts program execution and waits for user commands, the usability of such a debugger would be crippled heavily. The fact that runtime information about variables, functions, and values will have been erased from the program makes it well-nigh impossible to implement a functional REPL.

This issue is typically tackled by including debug information in the executable binary file that may allow the external debugger to understand the memory layout of the program and therefore to read and write values to variable locations and call functions available in the binary. However, the typical lack of a compiler inside the program still renders (re)defining functions or class definitions in the running program impossible, further hindering the ability to fix a running program—unless a compiler is made available to be loaded as an external module available from the debugger.

This situation is further confounded by the popularity of the preceding approach. Debuggers and compilers are often bolted on top of programs instead of comprising their internal parts; also, using debuggers and compilers in batch languages is usually a high-overhead undertaking with feedback loops that rarely run less than a second (while sub-second development feedback loops are "par for the course" in Lisp). This experience leaves many people with an impression that using compilers and debuggers—and, by extension, flexible condition systems—as an integral part of the whole programming life cycle, from development through to testing and deployment, is inconvenient, unnatural, and best avoided. This book boldly attempts to remedy this failure of imagination—a failure which also contributes to the paucity of knowledge about the Common Lisp condition system possessed even by many CL developers themselves.

The author argues that the advantages provided by contemporary processors and amounts of memory outweigh any advantages gained by erasing runtime data from the program. Such language design dooms programs that encounter unexpected error situations to a world of having no choice other than to crash and optionally produce dumps for further analysis—and analyzing dead remnants instead of a living program is an inferior choice in most practical situations. While the author of the book fully understands the reasoning behind the idea of "gotta go fast," he also urges programmers to consider the adage "the problem with being faster than light is that you can only live in darkness."

4.9 Debugging and encapsulation

The condition system makes it possible for programmers to craft additional "moving parts" for every programming module written in CL. While this allows for greater expressiveness and flexibility, it also means greater responsibility to document these aspects properly as parts of the *protocol* (or *interface*) that is used to communicate with the given programming module in order to ensure basic guarantees of encapsulation enforced by that module.

Just like a protocol usually describes the global variables, functions, classes, and methods that are used to communicate with a given programming module, it should also describe condition types whose instances are allowed to be signaled from inside that module, accessor functions that are usable for reading information from these conditions, and restarts that are established and available for invocation by handlers and user code provided into the module.

When a condition is signaled, that condition should contain all information required to parse and understand its meaning. It is good programming style to ensure that a condition is fully self-contained, that is, that the condition handler, by accessing the slots of a given condition object, has full information about what exactly caused a given condition; this good style helps to ensure that even if a condition object is returned outside its signaling point and therefore the stack information has been destroyed, a programmer can still inspect the condition object to learn more about what actually caused that condition to be signaled.

4.10 Debugging and introspection

The approach we have described earlier covers the *declared* part of a contract covering the condition system. However, the actual act and process of debugging a Lisp system makes use of one more piece of knowledge: that the CL programming language enforces no access control inside the Lisp image. Any notion of "private" parts of the system is purely notational: where a symbol named `foo:bar` is, by convention, meant to be publicly accessible, `foo::bar` is meant to be private, and the sometimes encountered `foo::%bar` or `foo::%bar%` strongly meant to be private and internal.

This language paradigm renders it trivial for a malicious actor with REPL access to abuse the system; it also lifts the responsibility for avoiding badly structured code from the compiler level to the code review and external static analysis tool level. However, this paradigm also combines very well with using the interactive debugger and inspection tools available in every CL image.

Even if a Lisp program has been designed such that it provides little or no debugging information in the signaled condition object, then the programmer can still use the system debugger and compiler to modify the running system, and they can still use the system inspector to probe the values of global variables, the values of dynamic variables at every point on the stack, and the values of function arguments at every point of the stack. By inspecting these values in turn, the programmer can read, modify, and recursively inspect list and array elements, values of class slots, and other properties that would have been impossible to access in other programming systems.

This abundance of information provides the programmer with unparalleled visibility into the state of the system while it is being debugged, which is a boon, especially in tough situations where the information gleaned from such introspection may be crucial for the programmer to resolve the erroneous state.

4.11 Unknown unknowns: Control flow is not exception handling

Explaining the benefits of the condition system can be troublesome because the illustrative examples that one would normally make suffer from the *curse of knowledge*: it is hard to come up with an example that is both simple enough to understand and complex enough properly to demonstrate the value of decoupling offered by the condition system. This difficulty comes about because the value of the restart subsystem comes into play when there is *not enough knowledge* at the call site to make a good decision, and a simple example would have to show a situation where we have full knowledge; otherwise, it risks being too generic to be understood.

For example, if we write that a condition system allows us to recover from a runtime type error, it's trivial to say, "oh, just add a type check for that `check-type` at the beginning of this function." If we write that a condition system allows us to recover from mismatched formats of some data, it's trivial to say "but you can just stick an `(if (old-format-p data) (process-old-data data) (process-new-data data))` there to solve it." If we write that the condition system allows us to handle malformed user-provided input, the answer is "why don't you validate it in the code before you use it?", et cetera, et cetera.

Obviously, these proposed solutions are going to work; adding a conditional check in the code of a function is fully equivalent to signaling a condition of some type, handling it, and restarting some process with modified data. However, in order to understand the pointlessness of these approaches, we should observe that they are not concerned about *exception handling*; they are concerned about modifications to the standard *control flow* of the program. They are not trying to fix the situation from the outside; they are trying to fix it from the inside. And fixing it from the inside is not always viable nor even possible at all.

This is because if one were capable of computing these situations up front and preparing the recovery strategies ahead of time, then one is *no longer doing exception handling*—one doesn't have unaccounted for, exceptional situations, and therefore has nothing to handle! Rather, one is doing *flow control*; these two terms—exception handling and flow control—have become wrongly synonymous in some contexts, especially since some popular languages, such as Python, indeed commonly use exception handling to perform flow control.

To summarize and paraphrase terms coined by Donald Rumsfeld: while control flow deals with *known knowns* and *known unknowns*, exception handling is much more dealing with *unknown unknowns*. Something catches fire, we don't know what it is, none of our programmed recovery mechanisms have taken care of that situation, and we nonetheless need a means of recovering gracefully from the situation. *This* is a true exceptional situation which requires a reaction. Recovery from such a situation is achieved first and foremost by allowing human intervention: the debugger is started, the stack information is preserved, so a programmer has all the information to try to debug the situation and, in effect, let the program continue execution. Some of these fixes can then be easily added to code in by means of defining new programmatic condition handlers (hooks, callbacks) and restarts (choices) in code.

In other words, the earlier examples assumed that one could easily couple the means of signaling an error situation with the means of handling it. This assumption is not necessarily true; let's imagine that we have just run into an error of some sort and need to perform an interactive hotfix of a running system. Fixing the issue properly would require modification to the piece of code that signaled the error, which would require digging into the relevant module, for which we might not even have source code access, or even if we do have access, the source code might be impenetrable by us, and/or we might not be able to afford debugging it on the spot.

It is likely that the module will need a patch developed for it (either by the author, by us, or by some other contributor). We now know that it has a fault, and we may also need to live with this fault for multiple days (or weeks!) before it is fixed in the upstream code repository and the new version is packaged and delivered. When we have caused error-handling information to be stored in the dynamic environment and have arranged for restart points to be sprayed across our stack, we have created a situation where the error can be handled both without recompiling the faulting module and without destroying the stack—computation can continue when we have provided a proper recovery routine. We are not getting crashes, and stacks are not getting completely unwound.

Therefore, it is possible to think of a condition system—a combination of conditions, handlers, and restarts—as an intricate system of callbacks and restart points. Every place that calls `signal` is a point that can execute arbitrary code, hooks, and callbacks; every place on the stack that accepts non-local returns is a potential place to which we can return in order to resume the computation. This, along with the fact that Lisp is an interactive system (indeed, a lot of Lisp's power simply disappears when it's used like Java or Rust or any other batch language), allows for a lot of potential choices of how to handle the *unknown unknown*—the situation that just caused our program to land in the debugger.

This intricate runtime system is the reason why we can fix the issue interactively, and possibly store this fix in a new condition handler that we then patch on top of our running program. Of course, this hotfix might (and should) then end up getting factored into the code properly. This turns what was once a toplevel condition handler, which processes some data and invokes some restarts, into a proper part of the code: that situation is no longer unexpected or exceptional; our code becomes prepared for it (or rather, we think that it is, and reality will validate it). The condition system has served its purpose by then; it helped us keep the system alive and fix the issue on the spot. The *unknown unknown* became a *known known* or, in the worst case, *known unknown*; exception handling became mere control flow. That situation was handled properly—it is no longer exceptional, neither for our program nor for us as the programmers.

There obviously exist alternative approaches of solving the preceding problem: for instance, one may decide to solve this issue by using a strong type system and defining different types (e.g., in Haskell) or classes (e.g., in Lisp) for the distinct forms that the data may take. If one can successfully use the "parse, don't validate" pattern commonly found in strictly typed strictly functional code, then one doesn't need as many exceptions because many exceptional situations become simply unrepresentable in code. Still, this approach is limited in power; such representation is only as powerful as the given type system itself, and creating an arbitrarily powerful *and* generic type system is equivalent to solving the halting problem and therefore unreachable in practice. (For readers interested in the topic, there is an important talk by Gary Bernhardt, named *Ideology*, that touches the issue of strength of type systems and the differences between dynamically and statically typed languages from the perspective of "unknown unknowns.")

4.12 Debugging with alien technology

Introducing the condition system to programmers can be a slow, tiring process for one more reason, which we touched on in the earlier chapters: many programmers are accustomed to everything that is available for their inspection being a core dump, a simple stack-unwinding exception system, or an external debugger attached to the program. In the first case, the program is already "dead," with all its network connections and OS context destroyed; in the second, the state of the stack is long gone by the time the exception handler is allowed to execute; in the third, we rarely are allowed to debug the program on site where it crashes, because the use of debuggers tends to be associated with high computational and cognitive overhead.

The approach demonstrated by the Common Lisp condition system is not just foreign to many programmers but is downright *alien*, meaning it needs to be understood (at best) from scratch, or (at worst) by deconstructing the entrenched thought patterns wrought by experience with old debugging styles, experience which in some cases goes back many decades already. The point is that, despite having roots reaching 60 years into the past, the idea of the condition system is still new to many domains of programming.

This gap in understanding is the greatest hurdle standing in the way of the Common Lisp condition system and, to some extent, CL itself nowadays. As long as the CL programming world remains relatively small, the majority of programmers will fail to see the practical benefits offered by its condition system, and these tools will miss a chance to impact the greater programming world in a positive manner (in fact, the programming world currently seems dominated by a polar opposite approach: handling errors by erasing faulting batch-compiled programs, along with their entire virtual machines (container images), in order to create new virtual machines with new program instances, even in the face of the massive resource waste that this trendy approach propagates).

However, the old adage “constant dropping polishes a stone” has been proven many times over, as it has been proven that the Lisp family of languages and CL as a member of that family have outlived many other programming language families and dialects which have come, shone for some time, and gone. Therefore, the author retains some hope that this treasure trove of ideas that is the condition system will help attract a fresh generation of coders into the CL fold in order to breathe new life into the CL programming world, thus transforming the popular perception of exception handling and debugging into an enlightened one.

We wholeheartedly hope that this book contributes to this transformation, even as current programming paradigms appear to be set in stone.

4.13 Acknowledgment

We hope that the insights garnered from the last chapter of the book and this appendix will nicely cap off the reader’s understanding of the Common Lisp condition system, including better context to understand how it may be implemented and be working under the hood. A good litmus test of our success will be the extent to which this new knowledge piques readers’ curiosities about Common Lisp itself enough to result in them ending up as new members of the Common Lisp community (and as a side effect of such a trend, we’d also be interested to see how the ideas behind the Common Lisp condition system might spread into other programming languages). Ultimately, such trends will enrich us all—the producers and consumers of the software which touches an ever-growing part of our everyday lives.

The author would like once again to thank all the members of the Lisp and Hacker News communities who have contributed to this book and its online appendix well before its release by openly discussing the condition system and exception handling in

general. These contributions included tough questions, provoked unexpectedly detailed answers, provided further insight into the sphere of exception handling and condition systems in and outside the CL world, and generously enriched multiple other parts of this book in ways that the single mind of the author could never have achieved alone and indeed had not even considered possible.

The reader may want to consult the "Hall of fame" section once again for detailed information about all people who contributed to the final quality of *The Common Lisp Condition System*. The author hopes that he has not forgotten to include anyone there, which—as sad and unintended as it is—is certainly possible, given the sheer number of people who helped during this book's development process.

Everyone, please accept my wholehearted thanks. I hope to see you on `#lisp` on the Freenode IRC network.

With these final words contained in the appendix, this book is now complete.

—Michał "phoe" Herda, September 2020

PS. Hope that I did a good job, Kent.