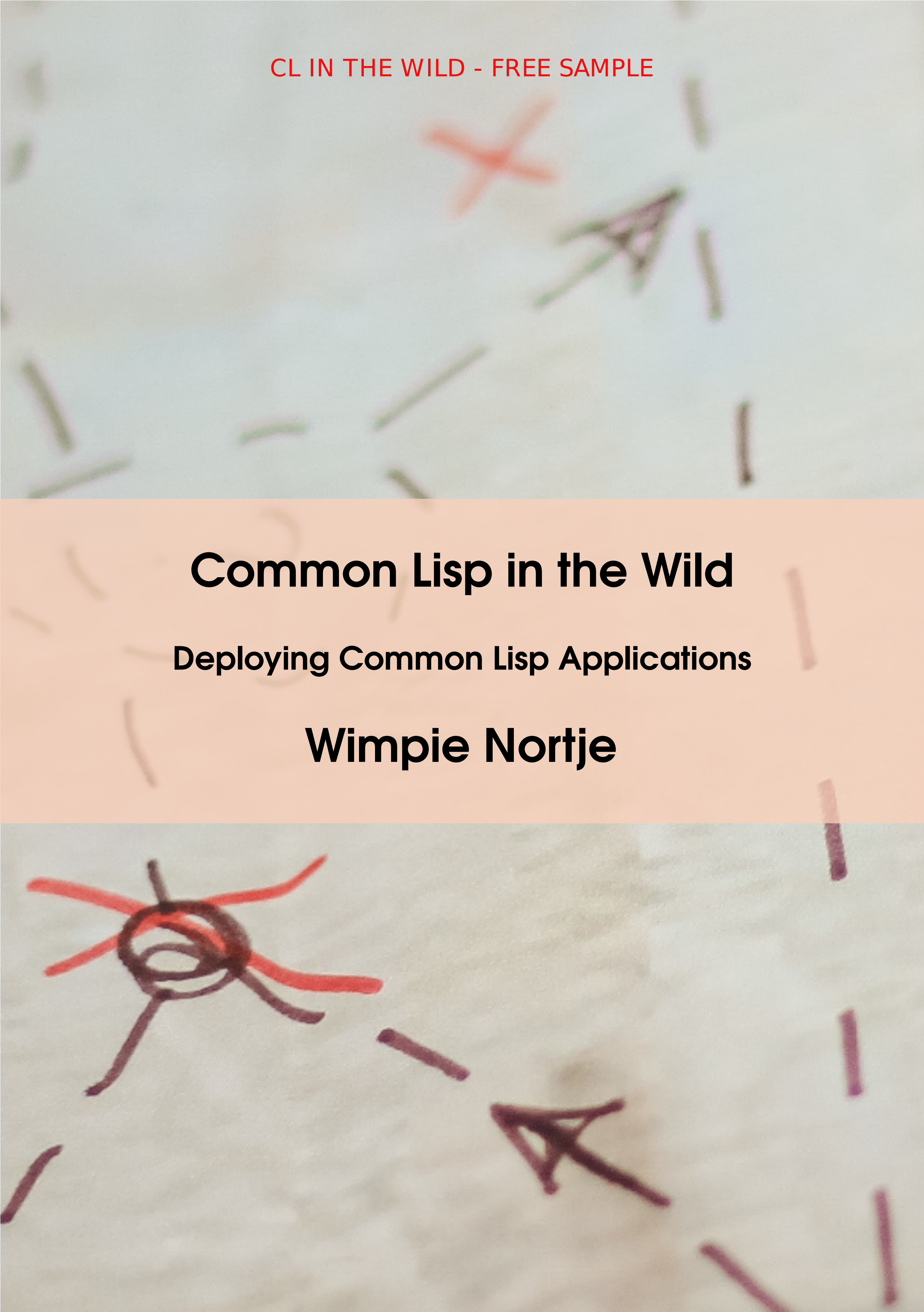

Common Lisp in the Wild

Deploying Common Lisp Applications

Wimpie Nortje

PUBLISHED BY WIMPIE NORTJE

WWW.DARKCHESTNUT.COM

Revision 2: March 27, 2017

1. Introduction

Common Lisp in the Wild is for people who want to create production software using Common Lisp. You may find the book useful if you need help deploying your first Common Lisp application or if you already use it in production and want to compare notes.

My goal is to get you from stuck to deployed in the shortest possible time. Except for the tidbits about portable binaries, everything in the book comes from my personal experience.

As with all things Lisp there are multiple options for every task. At the right time this is an empowering experience. At the wrong time it is daunting and paralyzing. The examples avoid selection paralysis by using the same tool for the same task every time. They are complete, to the point and build on each other without making conceptual jumps.

The tools were chosen to meet my needs for reproducible deployments. I did not evaluate many tools to find the perfect one but rather used the first one which worked. This leaves ample room for optimization if one needs to.

The toolchain I employ consists of the following:

Compiler SBCL and CCL. The examples only mention SBCL but they run as-is on CCL. Building CCL based binaries requires a CCL version of Buildapp. This requires some tinkering to get working which is not discussed in the book.

System definition ASDF

System management Quicklisp

System version management Qlot

Image generation Buildapp

Build system make

Basic Common Lisp knowledge is required to make sense of the book. You will also need a working development environment, including Quicklisp, to run the examples.

I hope you find the book useful and that it saves you some time and frustration.

7. Portability

Software is portable when it can be moved between different environments and still produce the desired results. Portability can refer to either the source code or the final binary. Section 7.2 briefly discusses binary portability but the bulk of the discussion is about source code portability.

Source code portability is a desirable characteristic because it enables the program to run on different types of computers without modifying the source code. It also allows the program to be compiled by multiple compilers without modification.

The first reason is important because it increases the program's target audience with very little effort. The second reason is important because the compiler can be chosen based on its strengths in any given situation. It is also a fact of life that products come and go. It is not prudent to lock a project into a specific compiler for no good reason.

There are other reasons why portability is desirable but these two reasons are important enough that it pays to keep you project portable, even when you don't think you need it.

7.1 Portable source code

Most applications can be written completely portable by paying attention to portability across operating systems, across compilers and across file locations. There are instances where one may need to consider lower level portability, like processor architecture, but they are beyond the scope of this book.

7.1.1 Operating system independence

Operating system independence is important when an application must be available on multiple operating systems. When the code is portable only one project needs to be maintained instead of a dedicated project for each operating system.

The easiest way to keep code OS independent is to use portable libraries which support all the necessary functions on all the target platforms. Many of the "trivial-..." libraries wrap OS specific details to provide a small set of functions with a single interface across all platforms.

Sometimes OS specific code is unavoidable. This code should be abstracted into an OS independent function which should have the same effect on all the supported operating systems. Listing 7.1 shows the principle.

Listing 7.1: Portability: Abstract OS specific code into functions.

```
#+windows
(defun os-dependent-calculation()
  (the-windows-way))

#+linux
(defun os-dependent-calculation()
  (the-linux-way))

#+osx
(defun os-dependent-calculation()
  (the-apple-way))

#-(or windows linux osx)
(error "Function 'os-dependent-calculation' is not
        yet implemented on this operating system.")

(defun main()
  (use-calculated-value (os-dependent-calculation)))
```

Special care must be taken with standard Common Lisp functions which vary between operating systems and compilers. Bugs due to these functions are especially hard to eradicate because the code tend to proliferate through a project and the differences between platforms are subtle. The main example for this is file system access. Use UIOP for everything related to pathnames.

Avoid all libraries which depend on particular operating system properties. They tend to provide capabilities which are not available on all operating systems. There are usually other methods or libraries that can achieve the same result in a cross-platform way. An example is Posix system calls.

The “windows”, “linux” and “osx” *FEATURES* used in listing 7.1 are sufficient when you stick with SBCL and CCL. Keep in mind that they are not completely portable accross all implementations though. The “trivial-features” library can be used to make your code portable across an even larger spectrum of implementations and OS’s.

Key points

- Use portability libraries for all interaction with the operating system.
- Abstract OS specific calls into a platform independent function.
- Use UIOP for everything related to pathnames.
- Never use libraries which depend on specific OS properties,

for example Posix compatibility.
- Use trivial-features to ensure that your feature checks are also portable.

7.1.2 Compiler independence

Compiler independent code saves you time and gives you options. There exists a number of mature Common Lisp compilers. All of them are very good in general and each excel in a different situation.

When your code is portable you can change your compiler any time during a project. You can use a fast compiling one during development and switch to a compiler which generates fast code for production, or you can select the compiler based on its support for the target operating system.

SBCL shows a disclaimer on Windows stating that it is unstable for multi-threaded applications. Keeping code compiler independent would allow the project to be developed on Linux in SBCL and deployed on Windows with CCL – without changing the source code.

Writing compiler independent code consist mostly of avoiding compiler specific extensions and libraries. SB-SYS and SB-EXT are examples of compiler specific libraries provided by SBCL. In CCL the "#_" reader macro is an example of an extension.

Most compiler specific code can be replaced by portable libraries. Should it really become necessary to use such code, it must be abstracted into a portable function. It is a good idea to ensure that the abstraction function will cause visible errors as early as possible when used on unsupported compilers. Listing 7.2 shows the principle.

Listing 7.2: Portability: Abstract compiler specific code into functions.

```
#+sbcl
(defun compiler-dependent-calculation()
  (the-sbcl-way))
```

```
#+ccl
(defun compiler-dependent-calculation()
  (the-ccl-way))

#-(or sbcl ccl)
(error "Function 'compiler-dependent-calculation' is
        not yet implemented on this compiler.")

(defun main()
  (use-calculated-value
   (compiler-dependent-calculation)))
```

Key points

- Avoid compiler specific extensions.
- Abstract compiler specific code into portable functions. Ensure it fails quickly and loudly on unsupported compilers.

7.1.3 Location independence

Location dependent standalone executables can not be deployed. They will only function on the machine where they were built. Location bound source distributions may work but they will hinder server administration. The issues addressed here mostly apply to standalone executables but addressing them in source code distributions can only be beneficial.

Programs often use the same directory for a particular task for long periods. User home directories, temporary directories and the application's installed location are all examples. It is almost instinct to use DEFVAR or DEFPARAMETER to declare values that hold the pathnames to these locations.

Listing 7.3: Location bound directory definition.

```
(defparameter *APP-CONFIG-DIR*
  (uiop:subpathname
   (user-homedir-pathname) ".myapp" :type :directory)
  "This is location bound because the value will be
   fixed at build time.")
```

Storing such a pathname as a constant value is a problem because the value will be fixed at build time. The value for the user's home directory will definitely be wrong because it will hold the home directory on the build machine. There is a high likelihood that the installation location will also be wrong because people often install programs in unexpected places. The temporary directory can also vary between installations due to system policy variance.

The solution for the changing pathames is to to calculate the pathname whenever this value is needed with a function.

Listing 7.4: Portable directory definition.

```
(defun app-config-dir ()
  "This is portable because the value is recalculated
   every time it is used."
  (uiop:subpathname
   (user-homedir-pathname) ".myapp" :type :directory))
```

When the directory under consideration is a "standard" system directory like the user's home directory or the global temporary directory, don't make assumptions about its value. The correct pathname for these "standard" system directories varies not only between different operating systems, but also between different versions of the same operating system. Especially on Windows. Getting the correct values for these directories is not a trivial task. Use the compiler's built-in functions for this purpose.

Listing 7.5 demonstrates functions for four regular cases. Note that it doubles as an abstraction for compiler specific code.

Listing 7.5: Portability: Calculate common directory pathnames.

```
;; User's home directory
(user-homedir-pathname)

;; General temporary directory
(uiop:temporary-directory)

;; Location of the .asd file (Source distributions)
(asdf:system-source-directory :my-project)

;; Location of the executable (Standalone binaries)
(defun executable-pathname ()
  #+sbcl sb-ext:*runtime-pathname*
  #+ccl (uiop:pathname-directory-pathname
         ccl:*heap-image-name*))
```

Every shared library dependency is another instance where the application is location dependent and which could make the program undeployable. Building standalone binaries which use shared libraries was discussed in depth in chapter 5.6. The suggestions mentioned there also apply to source distributions.

Key points

- Don't store pathnames with unchanging values in DEFVAR or DEFPARAMETER. The values are determined at build time, not run time.
- Calculate values for unchanging pathnames at run time with functions.
- Use the compiler's built-in functions to get pathnames for OS system directories.
- Ensure the suggestions in chapter 5.6 regarding shared libraries are implemented.

7.2 Portable binaries

The goal of source code portability is to write code which is compilable, without modification, on the widest possible range of targets. In contrast, the goal of binary portability is to build a standalone binary which is executable, without modification, on the widest possible range of targets.

Binary portability is important in grid-like computing environments. The SETI research project and the Steam gaming platform are well-known examples.

In this environment the executing machine can be of any hardware platform, running any version of any particular operating system. The machine's specifications are determined by the user. It is not viable to create an executable for every possible combination of hardware and software which may be available as executing machines. To reach the widest possible audience it is to the application provider's advantage if the same application binary can run on a variety of platforms.

Creating portable binaries is a specialized topic and is beyond the scope of this book. If you find it interesting you should consult resources dedicated to the topic. A few ideas which could help get you going:

- The distribution should include all of the application's dependencies. There should be no dependency on the operating system to provide libraries other than the most basic ones.
- During the build process, just before saving the Lisp image, update SB-SYS:*SHARED-OBJECTS* or CCL::*SHARED-LIBRARIES* to use absolute paths to the bundled shared libraries.