Clojure by Example

A Hands-on Introduction to Clojure

Author
Rüdiger Gad – r.c.g@gmx.de

Version 0.1.0 – November 11, 2012

Contents

About this Document

Clojure, in a nutshell, is a functional programming language that runs on-top of the Java Virtual Machine (JVM). In this guide I will try to introduce Clojure in a pragmatic, hands-on fashion. There will be some discussion of the background etc. but the focus is on working with Clojure and its tools.

I assume that you are reading this because you are interested in Clojure or functional programming in general. However, I like to make just some short remarks just in case you are not sure if you should read on. Personally, I like Clojure very much; it is a powerful language, offers great tools, and allows usage of a vast range of pre-existing libraries that help you to develop quickly and efficiently. Additionally, if you are, for now, only used to procedural (C or Turbo Pascal < 7.0) or object-oriented programming (C++, Java, etc.) the functional programming paradigm opens up an entirely new perspective on how computational or software engineering tasks can be solved. Using a Lisp (Clojure is a Lisp dialect.) you will find much less conventions built into the language that constrain you during development. While I really think Clojure is great I won't stress you too much and try to proceed right away. I hope you will discover the usefulness of Clojure and its power as you go on with this guide.

Throughout this guide, the Clojure specific terminology will be introduced step by step. In this guide, you will, amongst others, learn about things like: what it means to use a Lisp, what "homoiconic" means, functions being first-class, what the Read Eval Print Loop (REPL) is, and how to make use of all this. Besides these fundamentals, topics like editor support for writing Clojure code, test-driven development using Clojure, or automation for efficiently working with Clojure projects in day-to-day development will be covered as well.

Part I.

Preparation

1 Get Clojure

This is a hands-on guide. You are strongly encouraged to try all the examples yourself. So, the very first step is to actually get Clojure.

As Clojure runs on top of the JVM you need a working Java installation to actually work with Clojure. Since this guide is on Clojure and not on Java the details of installing Java are not covered here. Nowadays, most will have a working Java installation anyhow.

Depending on your setup, there are different ways for installing Clojure. In the following some of these ways are explained.

1.1 Installation via a Package Manager

If you are using a "common" Linux distribution like Debian, Ubuntu, Fedora, or Gentoo the convenient way to install Clojure is via the respective package manager. This way also all required dependencies (like Java) will be automatically installed if necessary. The following snippets, see Listings 1.1 to 1.3 on this page, show the corresponding commands for installing Clojure on some popular Linux distributions.

Listing 1.1: Installation of Clojure on Debian and Debian-based distributions like Ubuntu or Kubuntu

```
apt-get install clojure
```

Listing 1.2: Installation of Clojure on Fedora or Red Hat

```
yum install clojure
```

Listing 1.3: Installation of Clojure on Gentoo or Funtoo

```
emerge clojure
```

Note that all of these commands are to be run as "`root`" user.

1.2 Manual Installation

If, for whatever reason, you cannot or don't want to install Clojure via a package manager you can install it manually. If you already installed Clojure via a package manager as described in the previous section you can safely skip this section and go on in section 1.3. This approach also applies to operating systems

that do not offer centralized package management, like MacOS or Windows. In this case you need to take care of installing Java yourself, if it is not already installed.

The most recent Clojure version can be downloaded from `http://clojure.org/downloads`. I suggest to use the latest stable version. Simply download the `*.zip` file, extract it, and copy the contained `clojure-<VERSION>.jar` file somewhere on your computer. In case of Linux the directory "`/usr/local/lib`" could be a good place to put the JAR file. Note that you usually need `root` privileges to put a file in "`/usr/local/lib`", so, copy the file as `root` or via `sudo`.

As of the time of writing the latest stable version of Clojure is 1.4.0. Also note that we are not using the file ending with "`-slim.jar`" but use the "full version" instead.

Once you got the JAR file you can run Clojure, more precisely the Read Eval Print Loop (REPL), as shown in Listing 1.4.

Listing 1.4: Run Clojure from the Command Line

```
java -jar <PATH_TO_FILE>/clojure-<VERSION>.jar
```

For convenience, I suggest to create a script for running Clojure. This way, you do not need to memorize the full command line, including the location where you put the JAR file, but just need to know the name the script file.

If you put that file somewhere on your `$PATH` you can conveniently run that script from the command line like any other program. In case of Linux, the directory "`/usr/local/bin`" is usually a good place to put your custom scripts. Similarly to when copying the JAR file you will usually need `root` privileges to put files in "`/usr/local/bin`".

In Listing 1.5 and example of such a script, say "`/usr/local/bin/clojure`", for Linux is given. Do not forget to set the permissions accordingly such that common users can execute the script (see Listing 1.6).

Listing 1.5: Simple Script to Run Clojure

```
#!/bin/sh
java -jar <PATH_TO_FILE>/clojure-<VERSION>.jar
```

Listing 1.6: Setting Permissions for the Start-up Script

```
chmod 755 /usr/local/bin/clojure
```

With this script you should now be able to simply start the Clojure REPL by executing the script, e. g., by typing "`clojure`". We will soon improve this script in order to allow more convenient interaction with the REPL.

1.3 Test the Installation

At this stage you should have a working Clojure installation. To test this either run your custom start script as created in section 1.2 on the previous page or run the start script as supplied by your distribution, in case you installed Clojure via a package manager as described in section 1.1 on the preceding page. The Clojure start script is usually called "`clojure`".

Simply execute the respective script in a terminal. You should now see the Clojure REPL as shown in Listing 1.7.

Listing 1.7: First Start of the Clojure REPL

```
[rc@colin ~]$ clojure
Clojure 1.4.0
user=>
```

For now we are not doing anything with the REPL; it is sufficient that it is working. To exit the REPL for now type "`(System/exit 0)`" as shown in Listing 1.8.

Listing 1.8: Leave the REPL for now

```
[rc@colin ~]$ clojure
Clojure 1.4.0
user=> (System/exit 0)
[rc@colin ~]$
```

We will do the first steps in the REPL in section 3 on page 10. Before we do this we add a little it of "candy" for working more conveniently with the REPL (see section 2 on the following page).

2 Sweeten the REPL

At this stage you should already have a working Clojure installation. To be honest, the steps we do now are optional. Still, I highly recommend doing them now.

If you are impatient and want to hack right away you can skip this section and jump directly to section 3 on page 10. You can come back and do the steps described here at any time later on.

Right now, you can already launch a Clojure REPL. However, it is not very convenient to work with it as it is at the moment. We will change that here. We will modify the start script to allow much more convenient interaction with the REPL.

This modification will add a "history". With this history you can scroll back and forth in "old" things that you had already entered earlier. This is similar to what you might already be used to in a shell or terminal.

There are different options for realizing this. Depending on which option you choose, you will also be able to also search in your history using the key combination "CTRL+R", like in the Bourne-again Shell (bash). In the following I will describe two options:

- using `rlwrap`
- and using `jline`.

2.1 `rlwrap` Method

`rlwrap` is a wrapper for `readline`. This is the recommended method when you are using a Linux distribution as operating system. Reasons are that this is more powerful and it is easier to use as compared to using `jline`.

To use `rlwrap` two steps are necessary: installing `rlwrap` and modifying the Clojure start script.

2.1.1 Install `rlwrap`

First of all, you need to install `rlwrap` with your favorite package manager. Below Listings 2.1 to 2.3 on the following page exemplarily show the installation of `rlwrap` for some Linux distributions.

Listing 2.1: Installation of `rlwrap` on Debian and Debian-based distributions like Ubuntu or Kubuntu

```
apt-get install rlwrap
```

Listing 2.2: Installation of `rlwrap` on Fedora or Red Hat

```
yum install rlwrap
```

Listing 2.3: Installation of `rlwrap` on Gentoo or Funtoo

```
emerge rlwrap
```

2.1.2 Modify Clojure Start Script

Finally, the Clojure start script needs to be modified. This modification is very simple. It is sufficient to add "`rlwrap`" in front of the "`java`" call in the startup script.

In the case of the custom start script as created in section 1.2 on page 3 the new start script looks as shown in Listing 2.4.

Listing 2.4: Modified Start Script with `rlwrap`

```
#!/bin/sh
rlwrap java -jar <PATH_TO_FILE>/clojure-<VERSION>.jar
```

If Clojure was installed via the package manager as described in section 1.1 on page 3 there are two options: Firstly, a custom start script as described in section 1.2 on page 3 can be created or the existing start script can be modified.

When modifying the existing start script at first the start script must be located. You can locate the start script, e. .g, with `which`. In Listing 2.5 I show what it looks like on my computer.

Listing 2.5: Locating the Distribution Supplied Clojure Start Script

```
[rc@colin ~]$ which clojure
/bin/clojure
```

Now that we know the Clojure start script is "`/bin/clojure`" we can edit this file and add the `rlwrap` call. The modification is analogous to the modification of the custom start script as described above; simply "`rlwrap`" must be added in front of "`java`". I am using Fedora, the modified start script is shown in Listing 2.6.

Listing 2.6: Modified Clojure Start Script on Fedora 17

```
#!/bin/bash
exec rlwrap java ${JAVA_OPTS} -jar /usr/share/java/clojure.jar "$@"
```

2.2 `jline` Method

Similarly to `rlwrap`, `jline` is a wrapper that adds some convenience to textual input user interfaces. One difference is that `jline` is written in Java. This has the big advantage that it is operating system independent and also works with MacOS or Windows. On the other hand, `jline` is not as powerful as `rlwrap`. However, using `jline` is still a huge improvement as compared to using the REPL without it.

In the following, I will deliberately skip instructions for installing and using `jline` via a package manager. Reason for this is that I assume that if you are using a package manager you are going with the `rlwrap` method as described in section 2.1 on page 6.

Similarly to the `rlwrap` method, using jline requires the two steps of getting it and integrating it into the script.

2.2.1 Get `jline`

You can download `jline` from `http://sourceforge.net/projects/jline/files/`. Simply download the `*.zip` file, extract it, and store the `*.jar` file somewhere on your computer. Similarly to the manual Clojure installation method, a good place for putting the `*.jar` (on Linux) could be `/usr/loca/lib`.

2.2.2 Modify the Start Script

After `jline` had been downloaded and saved the start script needs to be modified. I will only cover modifying the custom start script here; As already stated, I assume that you are not going with this method when you used the package manager for installing Clojure and while, hence, favor the `rlwrap` approach. In Listing 2.7 the modified version of the script as created in section 1.2 on page 3 is shown.

Listing 2.7: Clojure Start Script Modified to Use `jline`

```
#!/bin/sh
CLASSPATH=$CLASSPATH:<PATH_TO_JLINE>/jline.jar:<PATH_TO_CLOJURE>/clojure-<VERSION>.jar
     java jline.ConsoleRunner clojure.main
```

Part II.

Introduction

3 First Steps in the REPL

In this chapter we will execute the first, simple Clojure code. We do this by interactively playing around and experimenting with the **Read Eval Print Loop** (**REPL**).

Up to now we only did some preparations. Now it is the time to get your hands on real Clojure code.

We will start very simple and will slowly increase the complexity. During this chapter we will introduce the very basics of Clojure like data types, how to "call" functions, basic data structures, how to define "variables", how to create our own functions, and will already mention some Clojure specific terminology.

Do not be disappointed, afraid, or confused if you don't grasp or memorize everything right away. I will try to repeat the important things later on in following chapters. Here, the primary intent is that you get started with actually playing around with Clojure. I just introduce some terminology already here so that you get used to this terminology. By the way, repetition is considered a good way for learning. ;)

3.1 Hello ~~World~~ REPL

As the very first step we will continue with a slightly modified tradition when learning programming languages. We will make the REPL say "Hello" to us.

So start the REPL by executing the respective start script as set up in chapter 1 on page 3. In the REPL type "`(println "Hello REPL!")`". The result should look like shown in Listing 3.1.

Listing 3.1: Say "Hello" REPL! – "Hello REPL!"

```
[rc@colin clojure-by-example]$ clojure
Clojure 1.4.0
user=> (println "Hello REPL!")
Hello REPL!
nil
user=>
```

What happened here? In the REPL you executed, more particularly evaluated, your first function ("`println`"). Here, we can make different noteworthy observations: firstly, how the REPL works and why it is called Read Eval Print Loop. Secondly, what the REPL displays. Additionally, you see that Clojure code is, most probably,

very different to what you might be used to (If you haven't done much with Lisp or one of its dialects yet.[1]).

3.2 How does the REPL work?

Shown in Listing 3.1 on the preceding page is what the output of evaluating your very first function in the REPL looks like. In the third line you can see the REPL prompt: "`user=>`". At this prompt you can enter arbitrary Clojure code.

1. In the first step, the REPL **reads** the code entered at the prompt (see Listing 3.1 on the previous page line 3).
2. Then it **evaluates** (In Clojure/Lisp terminology code is evaluated.) what had been read.
3. Afterwards it **prints** the results (see Listing 3.1 on the preceding page line 4 and 5).
4. Finally, it displays the prompt again and waits for input in order to perform the next **loop** iteration (see Listing 3.1 on the previous page).

So, this is, in a nutshell, the very simple and pragmatic reason why the Read Eval Print Loop or short "REPL" is called the way it is.

Please note that "reading" in this case is not what we humans usually associate with reading a text or in computing terms typically understand as "reading" a file from disk into memory. In fact during the so called **read phase** much more may happen than just "getting" code from the REPL prompt or from a Clojure file. The read phase is one of two phases during which code is processed in Clojure. The second such phase is the so called **evaluation phase**. The differentiation between these two phases will become important later when we start talking about what is known as "macros". For now, we simply ignore the distinction between those phases for the sake of simplicity and rather proceed with some more basics.

3.3 What does the REPL display?

If you have read the previous section the answer is very simple: the REPL prints out the result of evaluating the code put into it.

However, we have to be more accurate here. Clojure functions may "return" "something" as result of their evaluation. This result is one of the things printed at the REPL.

On the other hand, we may have console output (usually stdout or stderr) that is printed as well. In the above example we used the function "`println`" to write "Hello REPL!" to stdout. This console output is shown in line 4 of Listing 3.1 on the preceding page.

The actual "return value" is then printed below this output in line 5. Here the "returned" value is "**nil**". `nil` is a special value that can be compared to "`NULL`" or "`null`" in other languages like C++ or Java respectively.

[1] If you are new to Lisp-like languages be prepared, but not scared, to see lots of brackets. ;)

To get a better idea of this difference just enter some code that does not print anything but solely evaluates to a value. In order to try this we will simply add two numbers by entering "`(+ 19 23)`". The result is shown in Listing 3.2.

Listing 3.2: Simple Addition in the REPL

```
user=> (+ 19 23)
42
```

3.4 Clojure: A Lisp Dialect

Clojure is a **Lisp** dialect. This has certain implications on the way Clojure code is written and represented. In this section we will look at some of the basic implications connected with using a Lisp-like language like Clojure.

What you might have noticed first is that everything is written encapsulated within brackets "`(...)`". To understand this, a little bit of background about Lisp is required. Lisp is a shorthand for "**list processing**", and this is basically what all Lisp-like languages do: they process lists.

In Clojure, and many other Lisps, a **list** is denoted using round brackets: "`(<1st element> <2nd element> <3rd element> ...)`". So, when you entered "`(+ 19 23)`" at the REPL what you did was to enter a list with first element "`+`", second element "`19`", and third element "`23`".

At this point, it is important to know that there is a special convention in Clojure (or Lisps in general) that defines that lists are treated specially. This convention, essentially, says that the first element of a list is expected to be "something" that can be called or executed. That "something" may be a **function**, a **macro**, or a **special form**. For now, we will not look into the differences of these three; at this point, it is sufficient to know that the first element of a list is expected to be "callable" or "executable".

To cross-check the above statement and to make yourself a little bit more comfortable with this convention just try to enter a list that has a first element that cannot be evaluated; e. g., enter "`(3 5 9)` at the REPL and try to evaluate this. The result should look similar to what is shown in Listing 3.3.

Listing 3.3: Result of an Attempt to Evaluate a List with a Non-evaluable First Item

```
user=> (3 5 9)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn user/eval19 (
    NO_SOURCE_FILE:11)
user=>
```

Here, you can see that the attempt to evaluate a list that does not contain an item that can be executed in the very first position fails with an exception. More particularly, the exception states that and object of type "`java.lang.Long`" cannot be cast to "`clojure.lang.IFn`". This is because Clojure tries to cast the first element in the list to "`clojure.lang.IFn`" in order to call or execute it. However, the first element in this list is "`3`" which is of type "`java.lang.Long`". Thus, the exception is thrown as shown in Listing 3.3.

This special way lists are treated has some consequences when entering or coding lists. We will come back to this later when it gets important.

Another thing you can observe here is the notion of Clojure being **homoiconic**. Homoiconic, essentially, means that the source code is written in the data structures as provided by the language itself. So Clojure source code, basically, consists of Clojure data structures. In the very simple example here you could already see that the basic way to formulate expressions is done via lists. Later on we will also see how different data structures are used in different places. Being homoiconic is a very neat feature that, as we will see later, allows very powerful operations.

When you are new to Clojure you might be surprised by some of its conventions. However, keep in mind that there are very few conventions "built into" Clojure. If you compare this to a different language like C++ there are much more conventions built into the language which makes it much less flexible than Lisp-dialects like Clojure. Conventions defined in C++ are, e. g., keywords like "`class`", "`void`", "`int`", or "`new`" but also syntactical constructs like how class or method definitions are structured (using combinations of round and curly brackets with semi-colons as deliminators etc.).

Clojure, on the other hand, allows you to freely change it in nearly any way you want. You will be able to, essentially, create your own programming language that is tailored to fit the needs of a specific issue or task. This is what is a also known as **domain specific language** (**DSL**). However, for now, we do not go into such advanced topics like DSLs.

Yet another thing you might have noticed is that the "operation" is expected to be the first element of a list. Because Clojure expects the "operation" or "operator" to be the first element, it is also said to use "**prefix notation**". This is called "prefix notation" because the "operator" is "in front of" (which roughly translates to "pre" or "prefix" in Latin) the operands; e. g., "`(+ 1 2)`". Prefix notation might be a little different to what you are already used to from other languages like Python, C, C++, or Java, for at least mathematical operations. For mathematical expressions, these languages use what is referred to as "**infix notation**". The name of "infix notation" essentially describes that the operator is "in between" the operands; e. g., "`1 + 2`".

Prefix notation allows to have more than two operators per operation. E. g., with prefix notation you can do the following: "`(+ 1 2 3 4 5)`". With infix notation you would need to chain the operator like this: "`1 + 2 + 3 + 4 + 5`".

3.5 Help to Help Yourself

With this document I try to provide a guide to Clojure. However, you should be ready and prepared to search for answers on your own as well. Hence, I introduce some ways for finding help for Clojure.

3.5.1 Built-in Help

Within the REPL you can use a built-in interactive documentation look-up feature. This is roughly similar the interactive help features like in Python or R.

For this documentation look-up there exist two functions:

- **doc**

- and **find-doc**

With "`doc`" you can lookup the documentation for a particular function, macro, special form, etc. The syntax for using `doc` is as follows: `(doc <NAME>)`, with `<NAME>` being the entity you want to look up. An example for looking up a documentation is shown in Listing 3.4.

Listing 3.4: Documentation Look-up

```
user=> (doc +)
-------------------------
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
nil
user=>
```

In this example we looked up the documentation for the "+" function[2]. For now, you will most probably not understand everything that is printed there. Do not worry about this, as you learn more about Clojure you will step by step learn what all the things printed there mean. Here, it is sufficient for you to memorize that Clojure offers nice tools if you are in need for help. Note that when looking up macros and special forms, the output will mention that the particular entity is a macro or a special form.

With "`find-doc`" you can search in the documentation. In the simplest case, you can simply supply a string that will be searched for. An example of searching for a string is shown in Listing 3.5. Please note that, for the sake of brevity, not the full output is shown there. I put in "..." to indicate that things had been left out.

Listing 3.5: Search the Documentation for a String

```
user=> (find-doc "split")
-------------------------
clojure.core/partition-by
([f coll])
  Applies f to each value in coll, splitting it each time f returns
   a new value. Returns a lazy seq of partitions.
-------------------------
...
-------------------------
clojure.string/split-lines
([s])
  Splits s on \n or \r\n.
nil
user=>
```

A more sophisticate way to search the documentation is to use **regular expressions** (**regex**). `find-doc` also accepts regex and will then search for these. Assume you want to look up how to split strings you could, e.g., use something like this: "`(find-doc #"Split.*string")`". The result of this is shown in Listing 3.6.

Listing 3.6: Search the Documentation via Regex

```
user=> (find-doc #"Split.*string")
-------------------------
```

[2] Yes, in Clojure it is completely valid to name a function "+". Another interesting point might be that "+" is indeed a function and not something built into the language like is the case for C or Java.

```
clojure.string/split
([s re] [s re limit])
  Splits string on a regular expression. Optional argument limit is
  the maximum number of splits. Not lazy. Returns vector of the splits.
nil
user=>
```

Not that "#"something"" is the Clojure shorthand to create a regular expression. For now, you do not need to memorize this. I will remember you how to create regex once it becomes important again.

3.5.2 Clojure Cheat Sheet

The Clojure Cheat Sheet is intended for quickly looking up the most important information about Clojure. Essentially, it is a quick reference for the Clojure basics. Personally, I found it to be a invaluable tool for quickly looking things up.

You can browse the Clojure Cheat Sheet online at the Clojure homepage [3] or download it in PDF format from there: `https://github.com/jafingerhut/clojure-cheatsheets/blob/master/pdf/cheatsheet-usletter-color.pdf?raw=true`. I suggest to download the PDF version and store it somewhere on you computer, so you have it always with you.

3.5.3 Online API Documentation

On the Clojure homepage there is also a browsable API documentation. This documentation is very useful if you are searching for certain functionality that goes beyond what is covered in the Clojure Cheat Sheet.

Additionally, you will find links to the source code of all the functions or macros that make up the Clojure API there. This can be very handy if you like to peek into the code, e. g., to see how the real Clojure experts solve problems.

3.5.4 Books

There also exists quite a number of books about Clojure. Personally, I found "Clojure in Action" by Amit Rathore and "The Joy of Clojure" by Michael Fogus and Chris Houser, both available from Manning publications, very helpful for learning Clojure.

3.6 Bye REPL!

Last but not least you surely want to eventually close the REPL. Recall that Clojure is running in a Java Virtual Machine (JVM). So, for closing the REPL it is sufficient to shut the JVM down.

We can simply shut the JVM down via the static "`exit`" Method in the Java "`System`" class. To do this simply type "`(System/exit 0)`" in the REPL as shown in Listing 3.7 on the next page.

[3] `http://clojure.org/cheatsheet`

Listing 3.7: Quit the REPL

```
user=> (System/exit 0)
[rc@colin clojure-by-example]$
```

You did this already before in section 1.3 on page 4 when you tested your Clojure installation, but this time you know a little better what you are doing. Don't worry about too many details for now, we will cover them bit by bit.

Closing the REPL is presumably the very first time where you make use of Clojures (very great) Java interoperability features. In this example here you simply called a static method of a common Java class. Clojure offers many great ways for interoperating with Java.

Essentially, you can use all Java code and libraries inside Clojure [4]. In turn, you can also run Clojure code from within Java. But, as you might have guessed, we will cover more details later.

[4] I, e. g., use Clojures Java interoperability features in "cljNetPcap" (`https://github.com/ruedigergad/cljNetPcap`), a wrapper and convenience library for working with "jNetPcap" (`http://jnetpcap.com/`) from within Clojure.

4 Clojure Basics

Right now, you should have a working Clojure installation and know how to use the REPL. As with any other programming language, in order to get to know a programming language, you should have an at least basic knowledge of how to store things and how to do things.

4.1 Storing or Things

One of the most essential things you need to do in a programming language is to store data "somewhere". This "somewhere" is usually a variable. To do something similar like creating a variable, in Clojure, there is the concept of a **var**.

On the first glance, vars might look identical to variables in other languages. However, there are some differences that are important when using more advanced techniques. For the start, it is completely safe when you think of a var as variable. Still, I'll refer to these as "vars". You should get used to this terminology and keep in mind that vars are actually not the same as what you know as variables from other languages.

A var is essentially a **symbol** that references something else. This "something else" can be essentially everything in Clojure: basic types, complex types, or even functions. [1]

Vars are said to be "defined". This is done using the "`def`" special form. Listing 4.1 shows how a var can be defined and "accessed" at the REPL.

Listing 4.1: Experiments with Boolean Values at the REPL

```
user=> (def foo "bar")
#'user/foo
user=> foo
"bar"
```

What we actually did here is to **bind** the result of evaluating "`bar`" to the symbol "`foo`". As this is the initial binding[2] it is also called "**root binding**". We will later on pick up the terms "binding" and "root binding" again. For now, it is sufficient that you know where to put your data if you want to and how to use it.

[1] On our way on learning Clojure we will at many times come across the notion of functions being "first class". As we will discover, having **first class functions** is one thing that makes Lisp-like languages like Clojure so very powerful.

[2] You could also read this as initial definition, but "binding" is more "accurate".

When we "accessed" the data stored in our var what we actually did was to evaluate the var, or more precisely the symbol. When a var is evaluated the result is the object that is referenced by the symbol, in our case the string "`bar`".

4.2 Data Types

Introducing all the data types and their particular tidbits might be a little tiresome. Hence, I will try to keep this as much on the point as necessary. If you like more details you should experiment quite a lot with the REPL while reading this chapter.

When you experiment with the REPL you will probably find the "`type`" or "`class`" functions useful. With these functions you can get information about the type of an "object".

4.2.1 Basic Types

Clojure supports various basic types. If you already know a programming language many of these types should already be familiar for you. However, there may be also types you aren't used to (yet).

The classification of "basic" data type I do here is a little bit arbitrary. Still, I hope it makes somewhat sense to you. Basic Clojure data types are, e. g.:

- Boolean Values
- Numbers
  - Integer
  - Floating Point
  - Ratio
- Characters
- Strings
- Keywords

Boolean Values

Boolean values are probably the most simple data types in any programming language. In Clojure the two boolean values are represented as "**true**" and "**false**".

Connected to boolean values there is a very simple rule in Clojure: everything except **false** and **nil** is treated as `true`. So, the only values that are treated as false are `false` and `nil`. This is a little bit different than some other Lisps that also treat empty lists as `false`.

If you like to experiment at the REPL you can play around with different combinations of values. For this, you can use a simple statement like the one shown in Listing 4.2 on the following page.

Listing 4.2: Experiments with Boolean Values at the REPL

```
user=> (if true "true" "false")
"true"
user=> (if false "true" "false")
"false"
user=> (if 21 "true" "false")
"true"
user=> (if 0 "true" "false")
"true"
user=> (if "foo" "true" "false")
"true"
user=> (if nil "true" "false")
"false"
```

As a little explanation of what we did in this Listing. Essentially, we just entered a list like "`(if true "true" "false")`". This construct is evaluated as follows: the first element "**if**" is a special form that evaluates the second element of the list, in our case here "`true`". Depending on the result of this evaluation either the third element, in case the second element evaluated to "`true`", or the fourth element, in case the second element evaluated to "`false`", is evaluated and the result is "returned". Here in this example "`true`", by definition, evaluates to "`true`" and thus the third element ""`true`""[3] is evaluated and the result is "returned".

Numbers

Maybe the next more complex data type after boolean values are numeric data types. Clojure supports the "standard" integer and floating point data types as well as some more powerful representations.

"Under the hood", most numeric Clojure data types use some corresponding Java type or class. To get the class information of a data type you can use "`type`" or "`class`" as shown in Listing 4.3.

Listing 4.3: Types of Numeric Values

```
user=> (type 1)
java.lang.Long
user=> (type 2.3)
java.lang.Double
user=> (type 123456789123456789123456789)
clojure.lang.BigInt
```

Here you can see that Clojure really uses primarily the numerical Java data types like "**java.lang.Long**" and "**java.lang.Double**". More precisely, it uses the according Java classes instead of the simple Java data types.

Another noteworthy observation is that in the last statement the type is "**clojure.lang.BigInt**" or just "**BigInt**" in short. Clojure allows the seamless usage integer values of arbitrary size. The conversion from "`java.lang.Long`" to "`clojure.lang.BigInt`" is done automatically. Alternatively, it is possible to force the use of BigInt even for small values by appending an "`N`" like "`123N`".

Similarly to the "`123N`" notation, you can use "`4.56M`" to define a "**java.lang.BigDecimal**" or **BigDecimal** in short. This could be useful if you have special requirements with respect to floating point precision.

[3] This is a string. Note the quotation marks.

There is another data type in Clojure that is different from the ones above. This type is "**clojure.lang.Ratio**" or just **Ratio** in short. The ratio is used for representing fractions of individual integer values. An example is shown in Listing 4.4.

Listing 4.4: Ratio Data Type

```
user=> (/ 2 3)
2/3
user=> (type (/ 2 3))
clojure.lang.Ratio
user=> (/ 12 4)
3
user=> (type (/ 12 4))
java.lang.Long
user=> (/ 6 15)
2/5
```

In this example we used the "`/`" function for dividing two integer values. Note that the ratio type is only used when necessary as can be seen in the third and fourth experiment. Furthermore, note that fractions are automatically reduced as can be seen in the last test. With the ratio type it is possible to precisely represent fractions that could otherwise not 100% precisely represented with floating point numbers.

I suggest that you experiment at least a little bit at the REPL with the different numerical types and functions. Additional numerical functions you could use are "`+`", "`-`", "`*`", or "`mod`".

Characters

Characters are represented as "**java.lang.Character**" instances. To create such an instance the following notation is used: "
X". This creates the character "X"[4].

Strings

Similarly to most other types so far, Clojure strings are essentially Java strings: "**java.lang.String**". The straight forward way to define a string is to enclose it in quotation marks like this: "`"Some String"`".

Keywords

Keywords are a data type that is very unique to Lisp languages. To the best of my knowledge there is no equivalent to keywords in languages like C, C++, or Java.

Keywords are very special: keywords always evaluate to themselves. To get a feeling for this just try it yourself at the REPL as shown in Listing 4.5.

Listing 4.5: Keywords

```
user=> :foo
:foo
```

[4] Which can be pretty handy to mark a spot.

You can see that the keyword "`:foo`" evaluates to itself, namely "`:foo`". The convention for defining keywords is that they start with a colon, "`:`".

Later on we will also see that keywords can be used as functions. This can be very convenient when dealing with some complex data structures.

4.2.2 More Complex Data Structures

Here, I want to introduce the most important more complex data types that are available in Clojure. There are some even more sophisticated constructs that allow you to create your own custom data structures. For now, it is sufficient for us to explore the more complex data structures that are readily available straight away.

When you come from an object-oriented language like Java it could be a little difficult to think in these comparingly (compared to extensive class hierarchies) simple data structures. However, you will be surprised how powerful Clojures complex data structures are.

In the following we will cover the following data structures:

- lists,
- vectors,
- sets,
- and maps.

With these data structures you should be able to solve most computing/programming tasks.

Note that these data structures are immutable or "persistent". This means that, by default, a data structure cannot be changed after it had been created. Instead, when working with these data structures (like adding or removing elements), a new data structure that reflects the change is returned.

On the first glance this may sound very resource intensive. However, Clojure has some clever tricks to make this actually very efficient.

Lists

A List, in a nutshell, simply contains an arbitrary number of objects in a sequence. Unlike sets (see 4.2.2 on page 24), lists may contain the same object multiple times.

Recall that lists in Lisp are treated specially: the first element is expected to be something that can be called or executed. Because of this, lists as data structures cannot be simply entered "as-is". If you try to enter a list of integers like "`(1 2 3)`" as-is you will get an error as shown in Listing 4.6.

Listing 4.6: Error on Entering a List "as-is"

```
user=> (1 2 3)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn user/eval26 (
    NO_SOURCE_FILE:14)
user=>
```

However, you do not have to worry. There are ways how a list like this can still be entered.

One way is to use the "`list`" function. Listing 4.7 shows how this can be done.

Listing 4.7: Create a list.

```
user=> (list 1 2 3)
(1 2 3)
user=>
```

Another way is "**quoting**". With "quoting" you, basically, tell Clojure that it shall not treat a list specially, and thus try to execute the first element, but that it should take the list as-is instead.

You can quote in two ways: firstly using the **quote** special form, or, alternatively, with "`'`". Listing 4.8 shows how lists can be entered this way.

Listing 4.8: Enter lists using quoting.

```
user=> (quote (1 2 3))
(1 2 3)
user=> '(1 2 3)
(1 2 3)
user=> (type '(1 2 3))
clojure.lang.PersistentList
user=>
```

A list may contain different data types. Listing 4.9 shows this exemplarily.

Listing 4.9: A List Containing Different Data Types

```
user=> '(1 :keyword "String" \x)
(1 :keyword "String" \x)
user=>
```

Functions or accessing data stored in a list and working with lists are, e. g., "`nth`", "`peek`", "`pop`", or "`.indexOf`". Listing 4.10 shows these in action. Remember that we can create a var ("`def`"; bind to a symbol) for any entity in Clojure.

Listing 4.10: Work with a list.

```
user=> (def my-list '("one" "two" "three"))
#'user/my-list
user=> (nth my-list 1)
"two"
user=> (peek my-list)
"one"
user=> (pop my-list)
("two" "three")
user=> (.indexOf my-list "two")
1
user=>
```

Note that list indices start with "0".

Last but not least, adding elements to a list can be done with "`conj`" and "`cons`. This is shown in Listing 4.11.

Listing 4.11: Add things to a list.

```
user=> (def foo-list '("one" "two"))
#'user/foo-list
user=> (conj foo-list "three")
("three" "one" "two")
```

```
user=> (conj foo-list "three" "four")
("four" "three" "one" "two")
user=> (cons "three" foo-list)
("three" "one" "two")
user=>
```

Recall that Clojures core data structures are immutable. Hence, the “original” list is not changed by adding new elements. Instead the results of adding elements to a list are new lists that are, here in the example, printed to stdout.

Also note that when used with a list “`conj`” adds elements to the front of the list. The location where new elements are added depends on the type of collection data type that is passed to “`conj`”. Furthermore, note that “`conj`” allows to add multiple elements at once. “`cons`” on the other hand always adds one element to the front of a sequence data type.

If you want to quickly look-up a function for a specific purpose I suggest that you consult the Clojure Cheat Sheet[5].

Vectors

Similar to a list, a vector is a sequence of objects. One difference, however, is that the elements in a vector are indexed. This means that they can be quickly accessed by their index.

A vector can be created, e. g., with “`[]`” or “`vector`” as shown in Listing 4.12. In most Clojure code you will find/use the short-hand “`[]`”.

Listing 4.12: Create vectors.

```
user=> (vector 1 2 3)
[1 2 3]
user=> [1 2 3]
[1 2 3]
user=> [1 "two" :three]
[1 "two" :three]
user=>
```

One interesting property of vectors is that they are also functions on their own; a vector is a function of its indices. This is shown in Listing 4.13.

Listing 4.13: Vector as Function of its Indices

```
user=> (def my-vec ["one" "two" "three"])
#'user/my-vec
user=> (my-vec 0)
"one"
user=> (my-vec 2)
"three"
user=>
```

“`conj`” and “`cons`” are two ways for adding things to a vector. Note that when used with a vector, “`conj`” adds new elements to the end instead of the front, like is done when used with a list. “`cons`” on the other hand, works exactly the same way as with a list; it adds a single element to the front. The usage of these two functions is shown in Listing 4.14 on the following page.

[5] `http://clojure.org/cheatsheet`

Listing 4.14: Two Examplary Functions for Adding Things to a Vector

```
user=> (def bar-vec ["one" "two"])
#'user/bar-vec
user=> (conj bar-vec "three")
["one" "two" "three"]
user=> (conj bar-vec "three" "four")
["one" "two" "three" "four"]
user=> (cons "three" bar-vec)
("three" "one" "two")
user=>
```

Again, recall, that a vector is immutable. Hence, the original vector is not changed by adding elements to it.

I encourage you to have a look in the Clojure Cheat Sheet and play around with the different functions that are available for a vector.

Sets

A set may also contain multiple objects. However, unlike the previous two data structures (lists and vectors), sets cannot contain an object more than once.

The Clojure way to define sets is, e. g., "`hash-set`" or "`#{}`" as is shown in Listing 4.15. Usually, the short-hand version "`#{}`" is used most of the time.

Listing 4.15: Create sets.

```
user=> (hash-set 1 2 3)
#{1 2 3}
user=> #{1 2 3}
#{1 2 3}
user=> #{1 "two" :three}
#{1 "two" :three}
user=> #{1 "two" :three :three}
IllegalArgumentException Duplicate key: :three clojure.lang.PersistentHashSet.
    createWithCheck (PersistentHashSet.java:68)
user=>
```

Sets only contain a single instance of the same element; the attempt to create a set that contains the same element multiple times fails. This can be seen in the last example in the above listing.

Again, "`conj`" can be used to add elements to a set as shown in Listing 4.16. When an element is already contained in the set an unchanged version of the set is returned. Also note that, by default, there is no guarantee where elements are added to a set.

Listing 4.16: Add to a set.

```
user=> (def my-set #{"one" "two"})
#'user/my-set
user=> (conj my-set "three")
#{"one" "two" "three"}
user=> (conj my-set "three" "four")
#{"four" "one" "two" "three"}
user=> (conj my-set "two")
#{"one" "two"}
user=> (conj my-set "three" "four" 3 4 :kw :kx "abc" "def")
#{"abc" 3 4 "def" "four" "one" "two" :kw :kx "three"}
user=>
```

Similarly to vectors, sets are also functions. Sets are functions of their elements. If an element is contained in set it will be returned, otherwise `nil` is returned (see Listing 4.17).

Listing 4.17: Set as Function of its Elements

```
user=> (my-set "one")
"one"
user=> (my-set "fortytwo")
nil
user=>
```

The "`contains?`" function can be used to check whether a set contains a given item. "`contains?`" evaluates to either "`true`" or "`false`"; such functions are also referred to as **predicate functions**. In Clojure it is said to be "**idiomatic**" to append a question mark to predicate functions. With "idiomatic" it is meant that this no hard rule defined by the language but pure convention in order to ease the readability. In the literature you will encounter the term "idiomatic Clojure" quite frequently. Listing 4.18 shows "contains?" in action.

Listing 4.18: contains?

```
user=> (contains? my-set "one")
true
user=> (contains? my-set "fortytwo")
false
user=>
```

For more functions for working with sets I suggest to read the Clojure Cheat Sheet or the according API documentation. As usual, I encourage you to play around with these at the REPL.

Maps

A map is an associative memory. In a map "`values`" are assigned to "`keys`". For known `keys` the corresponding `values` can be easily looked up.

Ways for creating a map are, e. g., "`hash-map`" or "`{}`". As for the other data structures there are also other ways for doing this, however, I just exemplarily mention these two. Listing 4.19 shows these in action.

Listing 4.19: Create sets.

```
(user=> (hash-map :foo "foo" :bar "bar")
{:foo "foo", :bar "bar"}
user=> {:foo "foo" :bar "bar"}
{:foo "foo", :bar "bar"}
user=> {:foo "foo", :bar "bar"}
{:foo "foo", :bar "bar"}
user=> {:foo "foo", :bar "bar", "meaning-of-life" 42}
{"meaning-of-life" 42, :foo "foo", :bar "bar"}
user=>
```

The use of a comma is optional. In Clojure, commas are treated as white-space. However, some say it is easier to read when the different key/value tuples are separated by commas.

You might have already guesses it, like sets and vectors, maps are functions. This time maps are functions of their keys; they return the value for a given key or `nil`

if the key is not found (see Listing 4.20). In the last examples you can also see that keywords can be used as functions as well: when a keyword is used as function and a map is passed the keyword will be looked up in that map. If it is found the associated value is returned and nil otherwise.

Listing 4.20: Look things up in a map.

```
user=> (def my-map {:a "a" :b "b" "c" "see"})
#'user/my-map
user=> (my-map :a)
"a"
user=> (my-map :d)
nil
user=> (my-map "c")
"see"
user=> (my-map "Y")
nil
user=> (:b my-map)
"b"
user=> (:x my-map)
nil
user=>
```

The "`assoc`" function, as shown in Listing 4.21 is one way to add new key/value pairs to a map. If a key is already contained in the map, its value is set to the one supplied with assoc. Please have a look at the Clojure documentation like the Cheat Sheet for a full overview of functions that can be applied to maps.

Listing 4.21: Look things up in a map.

```
user=> (assoc my-map :d "d")
{"c" "zeh", :a "a", :b "b", :d "d"}
user=> (assoc my-map :b "bee")
{"c" "zeh", :a "a", :b "bee"}
user=>
```

4.3 "Make it so!"

So far, we just installed Clojure, learned a little bit about the REPL, looked into data types, and some complexer data structures. Now it is time to actually get things done!

One way to "do something" in Clojure is via functions. There are also other ways like macros but I consider those more advanced. Still, you will find out that functions are already very powerful on their own.

In this section we'll just look at functions. You will define your first functions, use them, and also learn a little bit more about what it means to use functions in Clojure.

4.3.1 Defining Functions

One way to "create" functions in Clojure is the "`fn`" special form. The syntax of "`fn`" is as follows: "`(fn [args] body)`". The "**body**" of a function is where the "things the function does" are defined. Listing 4.22 on the following page shows how "`fn`" is used.

Listing 4.22: Usage of fn

```
user=> (fn [] (println "My first function"))
#<user$eval7$fn__8 user$eval7$fn__8@6b915330>
user=>
```

So far so good, but how can we benefit from this? And what did actually happen?

What we did was to create a so called "**anonymous function**". An anonymous function simply is a function without a name. If you look at the listing again you can see that we simply used "`fn`" to create a function, and indeed we didn't do anything special to "give it a name".

Consequently, the "result" printed in the REPL, basically, is the string representation of a function in Clojure. This way we cannot do anything meaningful with that anonymous function right now. Later on, we will also find useful applications for such anonymous functions. However, for now, we start with simpler concepts, and thus, want to give our function a name such that we can call it via that name.

Recall that functions in Clojure are **first class**. This, essentially, means that we can handle functions like any other "entity", e. g., data types, of the language.

So, what would we need to do to give our anonymous function a name? In the preceding section we used "`def`" to create **vars** for storing data. If we combine the property of our functions being first class and the possibility to "store" something via the "`def`" special form the solution to this problem is easy, as is shown in Listing 4.23.

Listing 4.23: Use `def` and `fn` to Give a Function a Name

```
user=> (def foo (fn [] (println "Hello function!")))
#'user/foo
user=> foo
#<user$foo user$foo@23c24c1e>
user=>
```

To repeat what we did here:

- We created an anonymous function with "`fn`".
- A function in Clojure is "first class", i. e., an entity of the language not different than, e. g., data types or structures.
- We bound the anonymous function to the symbol "`foo`" with the "`def`" special form. That way we, essentially, gave the function a name.
- Last but not least, we checked, what we stored in the **bf** by accessing it the way we learned before. And indeed, we see an output similar as in the first example of an anonymous function; with the exception that we can see now that our function has a name.

I repeat this here in this very detail because this is an important property of Clojure that is very much different to what you may be used to from other, procedural or object-oriented languages. If you understand that functions are first class and what it means for functions to be first class you did a big step towards understanding Clojure and other Lisp-like languages.

Other important properties of functions being first class is that you can pass functions as parameters to other functions and even can return functions from functions. We will take a look into this later on.

By the way, do not confuse functions in Clojure with function pointers, e. g., in C. The property of functions being first class is much more powerful than what is possible with, e. g., function pointers.

4.3.2 "Calling" Functions

Right now, we know how to create functions and how to name them. But to really use them we also need to call them.

You should actually already know how to do this. Nonetheless, we will cover this here. So, don't worry, if you don't know how this is done right away.

Recall that a named function is just a var in which a function is "stored". Above we saw that when we just evaluate the var we get the function stored in the var but the function is not executed.

Recall as well that there is a special rule in Clojure (or "Lisps" in general) that says **lists** ("`(...)`") are special in the way that the first element of a list is expected to be something that can be called or executed.

So, if we combine these two properties, we can easily call our custom function. Listing 4.24 shows how this is done.

Listing 4.24: "Call" a Named Function

```
user=> (def foo (fn [] (println "Hello function!")))
#'user/foo
user=> (foo)
Hello function!
nil
user=>
```

In this example we simply called our named "`foo`" function we created.

Now, a question to you: Could we call an anonymous function as well? And, if "yes", how would we do this? To solve this, simply follow the few, simple rules you learned up to now. The solution is shown in Listing 4.25 on the next page.

Listing 4.25: “Call” an Anonymous Function

```
user=> ((fn [] (println "Hello function!")))
Hello function!
nil
user=>
```

That’s it. It’s as simple as this. And you will notice that it is following the same, simple rules all the time. Simply place the function as first element of a list. Thereby, it doesn’t matter if we use a named or anonymous function. Also, if our function does not have any parameters the list simply consists of a single argument,the function.

For now, we used very simple examples and using an anonymous function seems not to make sense. However, you will discover that there are good use-cases for anonymous functions.

4.3.3 Homoiconicity Refined

You learned that in Clojure code is data. You also learned that this property of the code of a language being written in the data structures of the language is called “**homoiconic**”.

Here, I just want to take a short look at this again. Up to now you only saw that **lists**, expressed using round brackets (“`(...)`”) are heavily used in Clojure.

If you were paying attention in the preceding chapters you should have noticed that another data structure had been used there to define code as well. Recall the syntax of a function definition: “`(fn [args] body)`”.

Here you can see that, additionally to a list, a vector, expressed via rectangular brackets (“`[...]`”), is used to define the function arguments.

4.3.4 Shortcuts

With Clojure it is possible to write very concise code. One way of how Clojure enables you to keep your code short and to the point are “shortcuts”. These shortcuts can be ways for doing multiple things at once or special syntax for frequently used things. Here I mention two “shortcuts” that ease the way functions are defined.

`defn`

Needing to type “`(def foo (fn ...))`” all the time is not too much code, especially as compared to other languages, but it may become tedious quickly. To safe us the worry, the Clojure developers created the `defn` macro.

With “`defn`” you can create a function and store it in a var in one step. The syntax of “`defn`” is similar to the syntax of “`fn`”: “`(defn foo [args] body)`”. In Listing 4.26 on the following page this is shown in action.

Listing 4.26: “defn” as Shortcut

```
user=> (defn [] (println "Hello function!")))
#'user/foo
user=> (foo)
Hello function!
nil
user=>
```

We will cover **macros** in more depth later. However, here you might be already interested in what the “defn” macro actually does.

Macros, in a nutshell, are functions that are used to write code. So, what “defn” does is, it creates the “(def foo (fn ...))” construct for you; i. e., a construct like “defn foo ...” will be expanded to “(def foo (fn ...))”. We can verify this in action when we use some more advanced techniques as shown in Listing 4.27.

Listing 4.27: “defn” transforms your code to “(def foo (fn ...))”.

```
user=> (macroexpand '(defn foo [] (println "Hello function!")))
(def foo (clojure.core/fn ([] (println "Hello function!"))))
user=>
```

I hope you are not confused right now. You should not try to understand everything here. What you should take away is that “defn” is some convenience utility that writes code for you and thus allows you, e. g., to keep your programs shorter. You can see this in the Listing 4.27: the “(defn foo ...)” construct is “transformed” to “(def foo (clojure.core/fn ...)”. Also note that this really transforms one bit of code into another bit of code; the result in the REPL is not some cryptic string representation of a function but is in fact Clojure code.

Later in this document you will learn how to write your own macros for doing such things. You will also learn how to do even much more sophisticated things with macros that open up a whole new world of how to “write” code. More precisely, you will learn how you can write programs that write your programs for you.

Anonymous Functions Abbreviated

You already learned that Clojure offers many ways to keep code short; e. g., when you learned about Clojures more complex data structures.

Anonymous functions are pretty often used, so it is not surprising that there is also an abbreviation for those. The shorthand for defining an anonymous function if: “#(...)”. Listing 4.28 exemplarily shows how this is done.

Listing 4.28: Shortcut for Creating Anonymous Functions

```
user=> #(println "Hello anonymous function shortcut!")
#<user$eval13$fn__14 user$eval13$fn__14@5803e54a>
user=> (#(println "Hello anonymous function shortcut!"))
Hello anonymous function shortcut!
nil
user=>
```

4.3.5 Passing Parameters

So far, we saw how to define functions, how to "call" them, and also had a look into the details under the hood, but we missed a very important functionality: we didn't use parameters yet. What we usually want to do in computing is to process data. In a nutshell, we usually have to pass data to functions in form of parameters.

We already briefly mentioned that parameters in a function definition are denoted as a vector. Now, we will see how we can make use of this and also have a look at some examples. For the sake of brevity we will directly use the "`defn`" macro instead of manually combining "`def`" and "`fn`".

We start by looking at the definitions of a function that accepts parameters. We will use a very simple example and create our own "`add`" function that takes two parameters and returns the sum. The definition of the "`add`" function is shown in Listing 4.29.

Listing 4.29: Defining a Function with Parameters

```
user=> (defn add [a b] (+ a b))
#'user/add
user=>
```

Here, we created the "`add`" function that accepts two parameters which are internally called "`a`" and "`b`". Then in the function we use the built-in "`+`" function to compute the sum of "`a`" and "`b`" which is then returned as result of our "`add`" implementation.

The usage of this function is similar to any other function, macro, or special form in Clojure. We, again, use the special rule that lists are treated speially, name the function as first list element and pass the parameters as second and third elements. This is shown in Listing 4.30.

Listing 4.30: Using our own Function with Parameters

```
user=> (add 1 2)
3
user=>
```

Arity

The term "**arity**" describes the number of parameters a functions accepts. In Clojure we can define so called "**multi-arity**" functions; functions that accept different numbers of parameters. E. g., recall our initial "infix-notation example" where we introduced the "+" function for the first time; we showed how we can pass different numbers of arguments to this function (for convenience see Listing 4.31).

Listing 4.31: The "+" function accepts different numbers of arguments.

```
user=> (+ 1)
1
user=> (+ 1 2)
3
user=> (+ 1 2 3 4 5 6)
21
user=>
```

So, how can we do this? Listing 4.32 shows our first attempt on creating a multi-arity version of our "`add`" function.

Listing 4.32: First Try on a Multi-arity Function

```
user=> (defn add
       ([] 0)
       ([a] a)
       ([a b] (+ a b))
       ([a b c] (add a (add b c))))
#'user/add
user=>
```

Note that I indented the code manually in order to make it easier to read. What this function definition does, is to create an "`add`" function that accepts either zero, one, two, or three parameters. As you see, the definition for each arity is a list that first takes a vector for the parameters and is then followed by the so called function "**body**". The "body" is, essentially, where the "things to be done" in a function are defined.

So, now, lets try to use our new, improved "`add`" function (see Listing 4.33

Listing 4.33: Using our First Multi-arity Function

```
user=> (add)
0
user=> (add 1)
1
user=> (add 1 2)
3
user=> (add 1 2 3)
6
user=> (add 1 2 3 4)
ArityException Wrong number of args (4) passed to: user$add clojure.lang.AFn.
    throwArity (AFn.java:437)
user=>
```

In the first tries, our function works as expected. I will not go into the details here, you should be able to see what's going on by comparing the examples with the definition. But when we try to pass a fourth parameter we miserably fail and get an "`ArityException`". This should be no surprise as we only defined the function for up to three parameters.

But remember the "+" function. There we could easily pass five parameters and (trust me here or try yourself) we could have passed any arbitrary number of parameters. How could we do this?

One, very long shot, could be to manually add definitions for even higher arities. This, however, not only would be extremely tedious, it would also not be possible to do this for all possible arities[6]. Furthermore, you would expect the folks at Clojure to be much more clever, and in fact they are.

The trick is that you can define functions to accept a **variable number of parameters**. This can be achieved with "**&**". "&" is a pretty neat trick. With "&" you can, basically, tell Clojure to put all remaining parameters (those not explicitly listed before the "&") into a list. The name of this list is then written behind the "&". However, you should at first experiment a bit with this to get used to it.

[6] Also note that the maximum arity of a function in Clojure is limited to 20.

In Listing 4.34 we, at first, define a function that accepts one “normal” parameter and takes then a parameter list using “&”. The function simply prints the first parameter and the argument list to “stdout”. Afterwards, we call this function with different numbers of arguments.

Listing 4.34: Testing Variable Length Parameters

```
user=> (defn test-arity [a & foo] (println "First parameter:" a "Remaining parameter
    list:" foo))
#'user/test-arity
user=> (test-arity 1)
First parameter: 1 Remaining parameter list: nil
nil
user=> (test-arity 1 2)
First parameter: 1 Remaining parameter list: (2)
nil
user=> (test-arity 1 2 3)
First parameter: 1 Remaining parameter list: (2 3)
nil
user=>
```

Here, you can see that if only a single parameter is passed, the list “foo” is “nil”. Later, when more parameters are passed you can see that these “remaining” parameters are put into the list named “foo”. This way a variable number (equal or bigger than the number of “normal” parameters) can be passed.

One way to exploit this feature is via recursive function calls. Thereby, the parameter list can be consumed step-by-step; with each recursive call the remaining list is reduced until all parameters had been processed. This is shown in Listing 4.35.

Listing 4.35: Recursively Processing Variable Length Parameters

```
user=> (defn test-arity-rec
       ([a] (println a))
       ([a & b] (test-arity-rec a) (apply test-arity-rec b)))
#'user/test-arity-rec
user=> (test-arity-rec 1)
1
nil
user=> (test-arity-rec 1 2)
1
2
nil
user=> (test-arity-rec 1 2 3)
1
2
3
nil
user=>
```

When called with a single parameter the function shown here simply prints that parameter. When more parameters are passed the first parameter is printed by calling the one-parameter version and then the rest of the parameters (the list “b”) is processed recursively as long as all parameters had been processed.

With this knowledge we can improve our “add” function from the beginning. A possible way of doing this is shown in Listing 4.36.

Listing 4.36: Improved “add” Function

```
user=> (defn add
       ([] 0)
```

```
       ([a] a)
       ([a b] (+ a b))
       ([a b & c] (apply add (add a b) c)))
#'user/add
user=> (add)
0
user=> (add 1)
1
user=> (add 1 2)
3
user=> (add 1 2 3)
6
user=> (add 1 2 3 4)
10
user=> (add 1 2 3 4 5)
15
user=>
```

The trick here is the last definition which enables the use of a variable number of parameters via "`& c`". What "`(apply add (add a b) c))`" does is this: it first adds "`a`" and "`b`" via the two parameter version of "`add`" and then calls "`add`" with the result and the remaining parameters in the list "`c`".

Passing Parameters to Anonymous Functions

Similarly to named functions you can also pass parameters to **anonymous functions**. One way is to use the "`fn`" special form and define parameters exactly the same way as we did above for "`defn`" (see Listing 4.37).

Listing 4.37: Anonymous Function and Parameters via "`fn`"

```
user=> ((fn [a] (println a)) 1)
1
nil
user=> ((fn [a b] (println a b)) 1 2)
1 2
nil
user=> ((fn [a b & c] (println a b c)) 1 2 3)
1 2 (3)
nil
user=> ((fn [a b & c] (println a b c)) 1 2 3 4)
1 2 (3 4)
nil
user=>
```

With the shortcut version of "`fn`" parameters are denoted using "**`%`**". With a single parameter using "`%`" and "`%1`" is equivalent. For more parameters you can simply use increasing numbers "`%2`", "`%3`", etc. Last but not least, "`%&`" is the syntax for variable length parameter lists. This all is shown in Listing 4.38.

Listing 4.38: Anonymous Function and Parameter via Shortcut

```
user=> (#(println %) 1)
1
nil
user=> (#(println %1) 1)
1
nil
user=> (#(println %1 %2) 1 2)
1 2
nil
user=> (#(println %1 %2 %&) 1 2 3)
```

```
1 2 (3)
nil
user=> (#(println %1 %2 %&) 1 2 3 4)
1 2 (3 4)
nil
user=>
```

4.3.6 Higher-order Functions

A "**higher-order function**", in a nutshell, is a function that takes another function as parameter and/or returns a function as result. This may, at first, appear strange to you when you are not used to functional programming, but this is a very powerful feature.

One common use case for higher-order functions is, e. g., to process sequences and apply an operation to its elements. Say, we have a vector of integers and want to double or square these. That's what the "`map`" function is for[7]. Listing 4.39 shows this in action.

Listing 4.39: Using the Higher-order function "`map`" to Process a Sequence

```
user=> (map #(* % 2) [1 2 3 4 5])
(2 4 6 8 10)
user=> (map #(* % %) [1 2 3 4 5])
(1 4 9 16 25)
user=>
```

[7] Do not confuse this with the "`hash-map`" function which creates the associative memory also called "map".

A Changelog

Table A.1.: Changelog

Date	Version	Remarks
2012-11-04	0.1.0	Initial Version

References

The references are split in two parts. First, the “classical” literature is given. Afterwards, references to web sites are listed.

Literature

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure*. Manning Publications Co., 2011.

[Hal09] Stuart Halloway. *Programming Clojure*. Pragmatic Booksheld, 2009.

[Rat12] Amit Rathore. *Clojure in Action – Elegant applications on the JVM*. Manning Publications Co., 2012.

Web Sites

[Hic12] Rich Hickey. Clojure - home. `http://clojure.org/`, 2012. last accessed 2012-11-04.

[jli12] Jline - jline. `http://jline.sourceforge.net/`, 2012. last accessed 2012-11-04.

[www12] rlwrap – freecode. `www.freecode.com/projects/rlwrap/`, 2012. last accessed 2012-11-04.

Index