Unit 0

Get Programming with Clojure

Welcome to Get Programming with Clojure!

In the current Unit, we will give you a taste of the Clojure syntax and explore together what specifically makes Clojure a fun language to learn and use and when it is most appropriate to use Clojure.

We will discuss the following topics:

- Lesson 1 introduces the Clojure syntax and its particular usage of the parenthesis
- Lesson 2 describes at a high level Clojure key features
- Lesson 3 discusses the use cases where it makes the most sense to use Clojure

The core of this Unit is **Lesson 2** where we explain what makes Clojure such a powerful language. We present several Clojure code snippets in order to make the ideas concrete. That is the reason why we present the basics of the Clojure syntax in **Lesson 1**.

After completing this Unit, I hope that you will be motivated and well prepared to start your journey into this wonderful language called Clojure.

More details about the Clojure syntax and Clojure installation instructions will come in **Unit 1** and **Unit 2**.

There are several elements that appear in throughout the book:

1. **Definition**: We define concepts in the context of their first usage. Reading the definition is mandatory for the understanding of the section where the definition appears.
2. **Keypoint**: We highlight the main ideas of the book as key points, like a core principle of the language or a best practice. The idea expose in a keypoint also appears in the context of the keypoint. Highlighting a keypoint serves two purposes: emphasizing the importance of the idea and allowing the reader to find it quickly.

3. **Discussion**: Some interesting questions that might occur in the mind of the reader are addressed in a discussion element. Discussions clarify some concepts but they are not mandatory for the understanding of the rest of the book.
4. **Extension**: Sometimes an advanced reader might notice something special or might be curious about some deep topic. The purpose of the Extension elements is to address those concerns. They are not mandatory at all.

Lesson 1

The Clojure syntax

Clojure is a LISP dialect. Lisp is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation.

As a LISP dialect, Clojure uses the LISP syntax and its fully parenthesized prefix notation. Let's see what it means.

There is a fundamental difference in the order of the parentheses between the LISP syntax and the syntax of most other languages. In most non-LISP languages, we usually have different syntaxes for different parts of the language while in Clojure all parts of the language follows the same syntax: the syntax of an s-expression (a shorthand for symbolic expression).

In this Lesson, we will illustrate the uniqueness of the Clojure syntax by focusing on function calls, arithmetic expressions and `if` expressions at a very high level. Those topics will be covered in much more details in **Unit 1** and **Unit 3**.

After reading this lesson, you will be able to:

- Describe the order of the elements in a Clojure expression
- Read function call expressions in Clojure
- Read arithmetic expressions in Clojure
- Read `if` expressions in Clojure

Table 1.1
This table summarizes the Clojure syntax for the expressions that will be covered in this Lesson

Expression	Example	Java-like syntax
Function call	`(foo 1 2)`	`foo(1, 2)`
Arithmetic operation	`(+ 2 3)`	`2 + 3`
Arithmetic comparison	`(> x 2)`	`x > 2`

if expression	`(if (> x 2)`
	`  3`
	`  5)`
		`if (x > 2) {`
		`  return 3;`
		`} else {`
		`  return 5;`
		`}`

Nested function call	`(foo 1` `  (bar 2 3))`	`foo(1, bar(2, 3))`
Nested arithmetic expression	`(* (+ 2 3)` `  (+ 5 7))`	`(2 + 3) * (5 + 7)`

1.1 Function calls

In most non LISP-based languages, in a function call, the function symbol precedes is followed by parentheses and inside the parentheses, the function arguments are separated by commas. Something like: `foo(1, 2)`.

When the arguments are themselves function calls, we have a nested function call like: `foo(1, bar(2, 3))`.

In Clojure, the function symbol is inside the parenthesis. It might feel a bit unnatural to you because you are so used to the other notation. You have probably already heard that in LISP, there are so many parentheses.

But if you think about it, you will see that this notation has the merit to be symmetric. It treats in the same way the function and its arguments: all go inside the parenthesis.

For a flat function call, the equivalent of `foo(1, 2)` is `(foo 1 2)`.

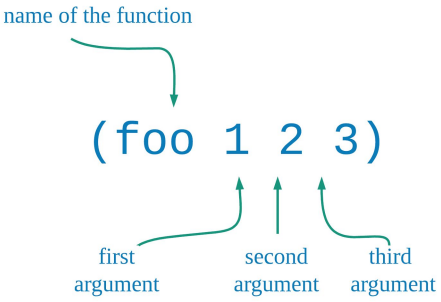

Figure 1.1
The structure of a function call illustrated by an example that involves a function named `foo`.

The syntactic rules for a function call in Clojure are:

1. A function call is enclosed into opening and closing parenthesis

2. First element inside the parentheses is the name of the function to call
3. Other elements inside the parentheses are the arguments to be passed to the function: the elements are separated by one or more whitespace characters. (Commas between the arguments are optional in Clojure, usually we don't use them.)
4. When one of the arguments is itself an expression, it follows the same syntactic rules

> **[EDITORIAL NOTE] Definition**: **Whitespace** means any character which are used for spacing, and have an empty representation. In the context of Clojure, it means spaces, tabs and end of line characters. A proper choice of whitespace leads to an indentation of the code that facilitates the reading of the code.

For a nested function call, the Clojure equivalent of `foo(1, bar(2, 3))` is `(foo 1 (bar 2 3))`.

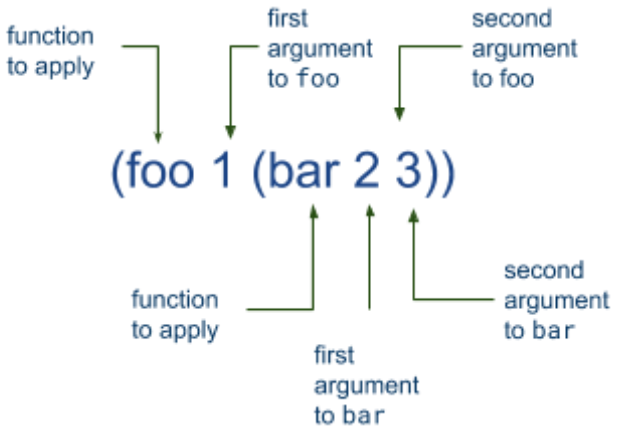

Fig 1.2
An example of a nested function call, the Clojure equivalent of `foo(1, bar(2, 3))` is `(foo 1 (bar 2 3))`.

Table 1.2

This table summarizes the translation between Java-style and Clojure syntax for flat and nested function calls

Java style	Clojure
foo(1,2)	(foo 1 2)
foo(1, bar(2, 3))	(foo 1 (bar 2 3))
foo(baz(4, 5), bar(2, 3))	(foo (baz 4 5) (bar 2 3))

As you can check by yourself, in the case of function calls (flat or nested), the number of opening and closing parenthesis is exactly the same in both syntaxes.

The higher number of parentheses in LISP and Clojure comes from the fact that this syntactic rule applies to all the expressions of the language: arithmetic operations function definitions, if statements etc..

1.2 Arithmetic expressions

Let's take a look at how an arithmetic expression looks like in a Java-style language. The arithmetic operator syntax differs from the function call syntax. Instead of being prepended to its arguments, like in a function call, the operator symbol comes between the numbers like this:

```
2 + 3
```

In Clojure, it will be `(+ 2 3)`.

The + symbol is just another function to be called. As such, it exactly follows the syntax of the function calls that we saw in the previous section.

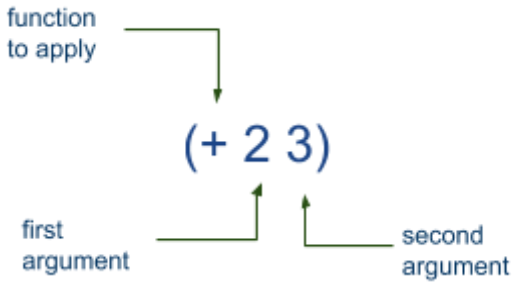

Figure 1.3
The structure of an addition with the + operation, illustrated by an example.

It is important to notice that the + function in Clojure can receive more than two arguments which is very handy and indeed makes the arithmetic expressions shorter than in other languages in some cases.

For instance, `2 + 3 + 4` would be `(+ 2 3 4)`.

And a nested arithmetic expression like `(2 *3) + (5 *7)` will be: `(+ (* 2 3) (* 5 7))`.

The same logic applies to arithmetic comparison operators. If we want to check whether the value of x is greater than 2, we write `(> x 2)` instead of `x > 2`.

[DEFINITION] form
An expression where an operation is applied is called a *form*. The name of the form is given by the name of the operation. For instance, an expression like `(+ 1 2)` that starts with the + symbol is called a "+ form".

Java style	Clojure form
2 + 3	`(+ 2 3)`
2 + 3 + 4	`(+ 2 3 4)`
(2*3) + (5*7)	`(+ (* 2 3)` `(* 5 7))`
x > 2	`(> x 2)`

Table 1.3
This table summarizes the translation between Java-style and Clojure syntax for flat and nested arithmetic expressions

1.3 `if` expressions

Now, let's take a look at how an `if` statement looks in a Java-style language. Let's write a piece of code that:

- compares x with 2
- Returns 3 in case x is greater than 2
- Returns 5 otherwise

```
if(x > 2) {
  return 3
} else {
  return 5
}
```

Please take a moment to notice that this expression doesn't follow at all the same syntax as the function call syntax. Let's decompose this expression:

- `if(x>2)` --- this part is similar to a function call
- `{ return 3}` --- here a new symbol appears: the curly braces
- `else` --- a symbol for defining what to do when the condition doesn't hold
- `{return 5}` --- curly braces again

In Java-style languages, the syntax of an `if` statement completely differs from the syntax of a function call. You are probably so used to it that you don't notice it anymore.

In Clojure, the previous `if` statement will be written as an if expression:

```
(if (> x 2) 3 5)
```

Please take a moment to observe the structure of this `if` expression and notice that it looks very similar to a function call. Indeed, in Clojure, `if` expressions follow the same syntactic rules as any other expressions in the language: an expression is made of an operator and optional arguments surrounded by parenthesis.

In the case of an `if` expression:

1. The first argument is the condition
2. The second argument is the value of the whole expression in case the condition holds
3. The third argument is the value of the whole expression in case the condition doesn't hold.

We can make this expression more readable by indenting it. Recall that we can add as many whitespace characters as we want without altering the meaning of the expression. The expression becomes:

```
(if (> x 2)
    3
    5)
```

We are not going to define formally the indentation rules in Clojure. Hopefully, you will get used to them naturally by reading the book.

In Clojure, there is no `else` keyword to introduce the value of the whole expression in case the condition doesn't hold. It makes the code more concise and once you get used to it, it is also more readable.

Like in many other languages, the `else` part of an `if` expression is optional: if we omit the third argument of an `if` expression it is replaced by the default `nil` value. For instance, the value of the following `if` expression:

```
<div class="runnable-client-clojure">
(if (> 0 2)
    3)
</div>
```

is `nil`.

Now, if we combine all what we learn we can write a complex nested `if` expression

```
(if (> x 2)
  3
  (if (< x 10)
      (+ x  20)
      (+ x 10)))
```

The occurrence of 3 closing parentheses at the end of the expression is an example of what make people say that in LISP there are too many parentheses.

In Java style syntax, it would have been:

```
if(x > 2){
    return 3
  } else {
    if(x < 10){
      return x + 20
    } else {
      return x + 10
    }
  }
```

Indeed, in this case in Clojure, we have 6 opening and closing parentheses exactly like in Java (2 opening and closing rounded parenthesis, 4 opening and closing curly braces).

Java	Clojure

`if(x > 2) {`
`  return 3`
`} else {`
`  return 5`
`}`
	`(if (> x 2)`
	`  3`
	`  5)`

`if(x > 2){`
`  return 3`
`} else {`
`  if(x < 10){`
`  return x + 20`
`  } else {`
`  return x + 10`
`  }`
`}`
	`(if (> x 2)`
	`  3`
	`  (if (< x 10)`
	`  (+ x 20)`
	`  (+ x 10)))`

Table 1.4

This table summarizes the translation between Java-style and Clojure syntax for flat and nested `if` statements: In Java-style languages we have indeed less parenthesis because the arithmetic expressions are not required to be wrapped into parenthesis. Also, we have different kind of parentheses: rounded parentheses and curly braces.

Finally, the fact that in Clojure, the convention is to close all the trailing parentheses on the same line give to some people this feeling that there are too many parenthesis.

My hope is that after having followed the first lessons of this book the Clojure syntax will feel much more natural to you. And hopefully, by the end of the book you will love the clojure syntax.

1.4 Summary

In this Lesson, we have introduced the basics of Clojure syntax and how it differs from other languages. We have seen that unlike other languages where we have a different syntax for functions, operators and statements, in Clojure the same syntax is applied to all parts of the language. We have explored examples of function calls, arithmetic expressions and `if` expressions.

This uniformity of the syntactic rules is what makes the community say that Clojure syntax is simple. At this moment, you might probably feel that this "simple" syntax feels a bit weird to you. In **Unit 1**, we will explore more in details the Clojure syntax and it will gradually feel more natural to you.

The high level knowledge of Clojure syntax that you acquired in this Lesson is enough in order to be able to understand the next Lesson that gives a preview of the key features of Clojure.

Lesson 2

A preview of Clojure key features

After reading this lesson, you will be able to:

- Describe the key features of Clojure at a high level
- Discuss the advantages of adopting Clojure

For many years, LISP languages have been considered as niche languages not widely used. They have been mostly used in the academics especially in Artificial intelligence research.

In contrast to most LISP languages, Clojure is a pragmatic programming language. It was designed in order to be used in the industry in order to address real life challenges. Nowadays, Clojure is widely used especially in the domain of Web development as we will see in Lesson 3.

In this lesson, we will unveil many aspects of this design decision and describe at a high level what are the key features of Clojure that make it a pragmatic language that allows developers to solve real-life challenges. Most of those key features will be explored in details later in the book. Don't expect to understand deeply at this stage the features exposed in this lesson. You can think of it as a preview of Clojure key features.

We could summarize the power of Clojure in a single sentence:

[EDITORIAL NOTE] Key point: Clojure is a pragmatic, data-driven language that offers high power of expression to its developers.

In this Lesson, we are going to explore 11 Clojure key features that can be grouped into three main themes:

1. **Pragmatic**
2. **Data driven**
3. **Power of expression**

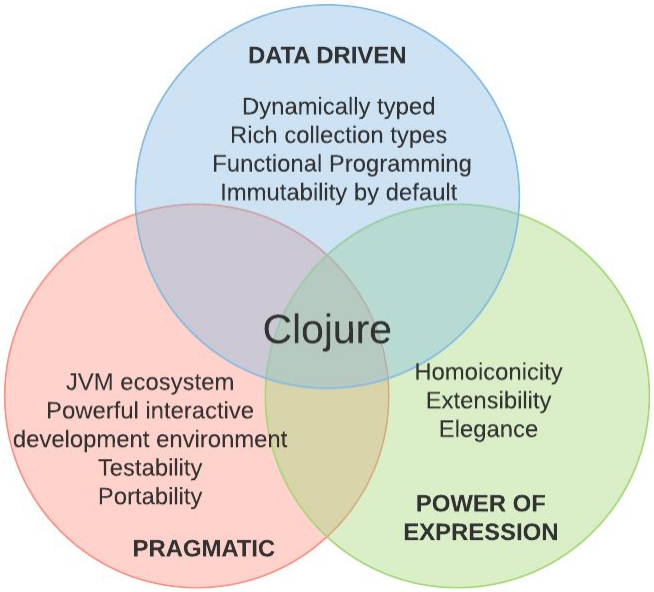

Figure 2.1 The themes of Clojure key features
Clojure key features can be grouped into three main themes: Pragmatism, Data driven and Power of expression. Clojure in the middle area of this Venn diagram symbolizes the fact that Clojure integrates those 3 themes: Clojure is a pragmatic, data-driven language that offers high power of expression to its developers.

2.1 Dynamically typed

Clojure is a dynamically typed programming language. It means that values have a type and their types are determined automatically. Therefore, the types don't have to be explicitly specified by the developer. It takes less effort to write the code and the code is easier to read as it is less verbose.

If you come from a dynamically typed language like Python, Javascript or Ruby, it probably sounds familiar to you. However, if you come from a statically typed typed language like Java or C#, it may feel a bit weird. Don't worry, we will explore in greater details the dynamically typed feature of Clojure in Lesson 11.

For now, let's have a quick look at of how variable definition works in Clojure.

In order to define a variable in Clojure, we use the `def` form that receives the name of the variable and the value to be assigned to the variable. The value could be of any type: a number, a string, a boolean or even a compound type like a vector.

For instance, we create a variable named `my-number` that holds the number `1.2` with:
```
(def my-number 1.2)
```

We create a variable named `my-string` that holds the string `"clojure is cool"` with:
```
(def my-string "clojure is cool")
```

We create a variable named `my-boolean` that holds the boolean `true` with:
```
(def my-boolean true)
```

We create a variable named `my-vector` that holds a vector with the numbers 1, 2 and 3 with:
```
(def my-vector [1 2 3])
```

In any case, the type of the value is determined automatically by Clojure.

Similarly, when we create a function, we don't have to specify the types of the function arguments. For example, here is an expression that defines a function named `add-numbers` that receives three numbers `a`, `b` and `c` as arguments and return their sum:

```
(defn add-numbers [a b c]
  (+ a b c))
```

Don't worry too much about the syntax of the defn form. It will be covered in Lesson 13. For now, it is enough to notice that the arguments of the function are declared by [a b c] and that there is not mention of the types of the arguments.

[Key point]: The dynamically typed nature of Clojure makes Clojure code concise and flexible.

2.2 Rich collection types

Clojure being a data driven language, collections play a very central role in Clojure.

Collections are data structures that allow values to be combined together like vectors (similar to dynamically sized arrays) and hash maps (similar to dictionaries). We use the terms maps and hash maps interchangeably.

In Clojure, manipulating data is really fun and productive because:

- The syntax for creating collections is clear and concise
- We can mix elements of different types in the same collection

- Clojure offers a rich set of functions for manipulating data inside collections
- Accessing data inside a collection is pleasant

For instance, we create a vector of numbers with the square brackets, where the elements are separated by a whitespace. Here is how a vector of numbers looks like:

```
[1 2 3 4 5]
```

The commas between the elements in a vector (or any other collection) are optional. The previous vector could also be written like this: `[1 2 3 4 5]`. It is quite common in the Clojure community to omit them. The advantage of using whitespace as a separator instead of comma is that you don't have to deal with the trailing comma dilema: whether or not to add a comma after the last element of a vector.

Because Clojure is dynamically typed, the elements of the collections are not required to be of the same type, like in Java or C#. For instance, we can create a vector with numbers and strings:

```
[1 "clojure" 3 "book"]
```

It is also very straightforward to create a nested vector. By nested vector, we mean a vector where some elements of the vector are themselves vectors:

```
[1 2 ["book" "get" "programming"] ]
```

- The first element of the vector is the number 1.
- The second element of the vector is the number 2.
- The third element of the vector is itself a vector made of three strings.

Clojure provides out of the box a very rich set of functions to manipulate collections.

For instance, look how simple it is to keep only the odd numbers out of a vector of numbers using `filter` and `odd?` functions:

```
(filter odd? [1 2 3 4 5 6 7 8 9])
```

The result is: `(1 3 5 7 9)`

And if you want to combine together the elements of two vectors, you use `interleave` like this:

```
(interleave ["a" "b" "c"] [1 2 3])
```

The result is: `(:a 1 :b 2 :c 3)`

For incrementing all the numbers inside a vector, you use the `mapv` and the `inc` functions, and the code is as short as:

```
(mapv inc [1 2 3])
```

The result is: `[2 3 4]`

Accessing data in Clojure is really a pleasant experience for the programmer: We use `get` to retrieves the value associated to a key in a hash map. And we use get-in to access a nested key. The cool thing with get-in is that, we can access a nested key in a hash map without having to check if all the intermediate keys indeed exist in the collection. If one of the keys along the path doesn't exist, `get-in` returns `nil`.

For example, accessing the `a.b.c` nested key of the empty object returns `nil`:

```
<div class="runnable-client-clojure">
(get-in {} ["a" "b" "c"])
</div>
```

The result is: `nil`

In other languages, the equivalent code would raise an exception and it is quite cumbersome for the developer to include code that deals with those exceptions.

2.3 Functional Programming

In Clojure, functions are first class citizens. It means that we can manipulate functions with the same freedom as we manipulate data.

It gives to the developer a very high power of expression in a very concise syntax.

[EDITORIAL NOTE] Key point: This section is only a high level preview of Functional Programming. We will explore in detail through the book the different ways of manipulating functions in Clojure.

In this section, we will give a short preview of how Clojure deals with 4 common patterns of functional programming:

1. Functions as arguments
2. Functions as return values
3. Anonymous functions
4. Functions in collections

2.3.1 Functions as arguments

When manipulating data, we often need to pass pieces of data manipulation logic across different parts of the system. In Clojure, the data manipulation logic is usually packaged in functions that are passed as arguments to other functions.

For instance, if we want to increment by one the numbers of a vector, we use the `mapv` function and pass it as an argument a function that increments by one a single number. Clojure provides such a function, it is called `inc`.

`mapv` receives a function `f` and a vector `v` and returns a new vector where `f` is applied to each element of `v`.

The code looks like this:

```
<div class="runnable-client-clojure">
(mapv inc [1 2 3])
</div>
```

The result is: `[2 3 4]`

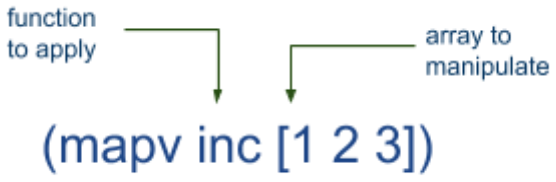

Figure 2.2
The structure of a function call where the first argument is itself a function illustrated by an example that involves `mapv` and `inc` functions. Here we apply the `inc` function to each element of the vector. The result is the vector `[2 3 4]`.

We will cover many other aspects of `mapv` and other functions that receive functions as arguments in Unit 10.

2.3.2 Functions that return a function

Sometimes, we want to create a new behaviour based on an existing behaviour with slight modifications. In Clojure, we can easily create functions that return a function.

One common example of a function that returns a function is the function named `partial`.

`partial` receives:

1. a function `f` (that receives `n` arguments)

2. an argument `my-arg`
and returns a new function `g` that receives `n-1` arguments and does exactly the same as `f` – where the first argument is always `my-arg`.

`(g a b c)` does exactly the same as `(f my-arg a b c)`.

This definition is a bit hard to grasp. Let's look at an example to clarify the meaning of `partial`. For instance, with the help of `partial`, we can create a function named add-42, out of the + function, where the first argument is fixed to 42.

```
<div class="runnable-client-clojure">
(defn add-42 (partial + 42))
</div>
```

We can now call add-42 like any other function:

```
<div class="runnable-client-clojure">
(add-42 31)
</div>
```

The result is: 73

One of the use case for creating functions through other functions is when we want to apply a simple piece of logic to the elements of a vector, usually with `mapv`. For instance, it takes a short one liner to increment by 42 the numbers of a vector:

```
<div class="runnable-client-clojure">
(mapv add-42 [1 2 3])
</div>
```

The result is: `[43 44 45]`

Notice that in that specific case, we didn't even need to give a name to the partial function. This is a general concept in functional programming: functions are not required to have a name, they can be anonymous.

2.3.3 Anonymous functions

Sometimes a function is so temporary that it doesn't make sense to give it a name. A function that has no name is called an anonymous function. You can think of anonymous functions as functions that are created "on the fly".

In a similar way to what we saw in the previous section, one of the use case for anonymous functions is when we want to apply a simple piece of logic to the elements of a vector, usually with `mapv`.

Let's say that we want to increment by 42 the numbers of a vector using `mapv`. Without using anonymous functions, we have to do two things:

1. Create a function with a name that increments by 42 a number
2. Pass this function to mapv

```
<div class="runnable-client-clojure">
(defn increment-by-42 [x]
   (+ x 42)
(mapv increment-by-42 [1 2 3])
</div>
```

The result is: `[43 44 45]`

If the the `increment-by-42` function has no other usage in our code, it makes sense to shorten the code by creating the function on the fly without even having to give it a name. The two expressions can be consolidated in a single expression that passes an anonymous function to `mapv`.

Usually, we create an anonymous function using the `fn` form that is similar to the `defn` form except that it doesn't receive a name. Let's write an anonymous function that increments a number by 42:

```
(fn [x]
   (+ x 42)
```

And here is how we increment by 42 the elements of a vector using `mapv` and an anonymous function in a single expression:

```
<div class="runnable-client-clojure">
(mapv (fn [x] (+ x 42)) [1 2 3])
</div>
```
The result is: `[43 44 45]`

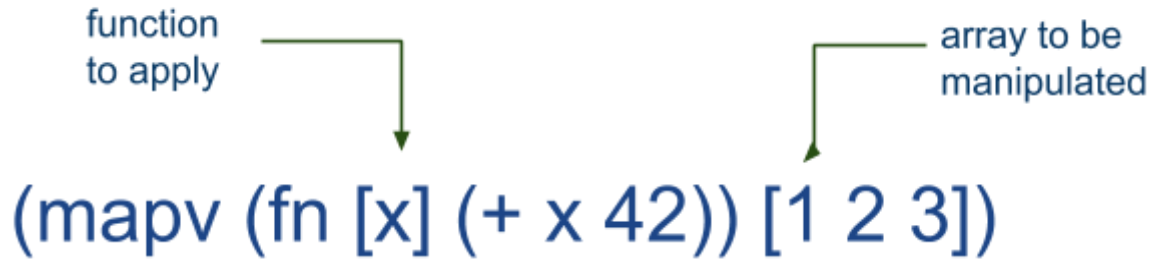

Figure 2.3
An example of using mapv and an anonymous function: here we increment by 42 the elements of the vector `[1 2 3]`, using `mapv` and an anonymous function in a single expression. The result is: `[43 44 45]`.

Anonymous function is a deep topic of functional programming and it will be covered in details in Unit 8.

2.3.4 Collection of functions

In Clojure, we can create collections where some or all of the element are functions. For instance, we can collect in a vector all the steps for processing an order in an online shop.

Let's say that in our program, we have three functions for the three processing steps: `register`, `checkout` and `pay`. Somewhere in our code those 3 functions are defined.

```
(defn register ...)
(defn checkout...)
(defn pay ...)
```

We can collect those functions in a vector `[register checkout pay]` and create a variable named `processing-steps` that holds this vector:

```
<div class="runnable-client-clojure">
(def processing-steps [register checkout pay]
```

Because functions are first class citizens in Clojure, we can manipulate a vector that contains functions in the exact same way to we manipulate a vector of data.

For instance, we can count the elements of the vector of functions like we count the elements of a vector of numbers using the `count` function:

```
<div class="runnable-client-clojure">
(count [1 2 6 10])
</div>
```
The result is: `4`

```
<div class="runnable-client-clojure">
(count [register checkout pay])
</div>
```

The result is: `3`

In Clojure, the elements of a vector don't need to be of the same type. Therefore, we can even mix data and functions inside the same vector. For instance, we can create another vector that contains both the processing steps and the max time allowed for each step, simply like this:

```
<div class="runnable-client-clojure">
(def processing-steps-and-max-time [register 100 checkout 200 pay 400]
</div>
```

We can then count the elements of this vector, with the count function:
```
<div class="runnable-client-clojure">
(count processing-steps-and-max-time)
</div>
```
The result is: `6`

Let's conclude this section with a table that summarizes at a very high level the 4 functional programming patterns that we introduced in this section.

Pattern	example	Clojure expression
Functions as arguments	Increment the values of the vector `[1 2 3]`	`(mapv inc [1 2 3])`
Functions as return values	Create a function that adds 42.	`(partial + 42)`
Anonymous functions	Transform the values of the vector `[1 2 3]` with custom function that adds 42.	`(mapv (fn [x] (+ x 42)[1 2 3])`
Functions in collections	Create a vector whose elements are functions: the `inc` function, the + function and the `*` function.	`[inc + *]`

Table 2.1
The different patterns of functional programming that are covered in this section. Each pattern is illustrated by an abstract example and a concrete Clojure expression.

2.4 Homoiconicity

2.4.1 What is homoiconicity?

In Clojure, the structure of the code is the same as the structure of the data. Basically, this structure is the structure of an expression: it is called an S-expression (symbolic expression).

Let's explore for instance how a list is represented. In Clojure, a list is represented with opening and ending rounded parentheses and inside them, the elements of the lists are separated with whitespace.

For instance the list of the elements 1, 2 and 3 is represented as: `(1 2 3)`.

The elements of a list are not required to be of the same type: for instance we could have a list made of 1 2 3 and "c". It would be represented as: `(1 2 3 "c")`.

Let's take a look at a more interesting example, a list made of functions and numbers, like the list made of the function +, the number 1 and the number 2. This list would be represented as `(+ 1 2)`.

Now come the homoiconicity into action. The same expression can be interpreted either as code or as data. Indeed, there are two ways to look at `(+ 1 2)`.

On one hand, `(+ 1 2)` is a list made of 3 elements: the symbol +, the number 1 and the number 2. On the other hand, as we have seen in Lesson 1, `(+ 1 2)` is a valid Clojure piece of code that means: the application of the + function to the arguments 1 and 2.

2.4.2 Why Homoiconicity is important?

The structure of a Clojure program is simple in the sense that only a small number of rules is required to understand its syntax. As a consequence:

1. As a developer, once you get used to it, it is simple to read clojure code
2. For development environments, it is very easy to do all kinds of smart indentation and code structure validation
3. Inside an editor, the code can be traversed like a tree. For instance, we can move to the next function definition in a clojure file is as simple as moving to the next element of a collection.
4. It is possible to comment out a single expression whereas usually, in other programming languages, the comments are either at the level of the lines or need to be explicitly terminated by the developer.
5. As a developer, you can write code that manipulates code. Welcome to the world of macros. Because the structure of the code is so simple, it is relatively easy for instance to write a macro that receives a function definition as an argument and insert at the beginning of the function a call to a logger. All your macro has to do is to manipulate lists. Many macros are already provided by Clojure out of the box.

We will elaborate on the advantages of homoiconicity through the book.

2.4.3 Different kinds of parentheses

In the original LISP, there were only rounded parentheses. In Clojure, we also have squared and curly parenthesis - both for the data and for the code. Having different kinds of parentheses makes the code more concise and increases the power of expression of the developers.

For example, in the original LISP, one had to write `(vector 1 2 3)` to create a vector with elements 1, 2 and 3 while in Clojure, one can write `[1 2 3]`.

The different kinds of parentheses have a different meaning in the data realm and in the code realm.

In the data realm:

- Rounded parentheses are for lists like `(1 2 3)`, a list of 3 elements 1, 2 and 3.
- Squared parentheses are for vectors like `[1 2 3 4]`, a vector of 4 elements: 1, 2, 3 and 4
- Curly braces are for maps `{“age” 20 “name” “John”}`, a hash map with two key-value pairs: the first one with key `“age”` and value `20` and the second one with key `“name”` and value `“John”`.

In the code realm:

- rounded parentheses are for function application.
- Squared parentheses are mainly for local variables assignment (will be covered in e Unit 6), function arguments declaration (will be covered in Unit 3).
- Curly braces are for destructuring a hash map into its key value pairs (will be covered in Unit 7).

It is not important at the moment to understand the details the meaning of each kind of parenthesis. The important point is:

[KEY POINT] Having different kinds of parentheses makes the code more concise and increases the power of expression of the developers

Type of parenthesis	Data realm	Data example	Code realm	Code example
Rounded parenthesis	lists	`(1 2 3)`	function application	`(+ 1 2 3)`
Squared brackets	vectors	`[1 2 3]`	local variables assignment, function arguments declaration	`(defn foo [x] x)`
Curly braces	maps	`{“age” 20 “name” “John”}`	destructuring	`(let [{name "name"} {"age" 20 "name" "John"}] name)`

Table 2.2
In Clojure, there are three kind of parenthesis: rounded parenthesis, squared brackets and curly braces . The meaning of the parenthesis is different in the data realm and in the code realm.

2.5 Immutability by default

In Clojure, the default way to manipulate data is the immutable way which means that data cannot be changed by the program.

Immutability is highly valuable because it is much easier for a programmer to understand and debug a program where it is guaranteed that the data is not modified. Many bugs that use to happen frequently in other programming languages are just not happening in Clojure because of the immutability.

In order to understand what immutability is, we need first to take a deeper look at how most programming languages handle change in data.

In most programming languages, the programmer can mutate a collection or an instance of a class. In other words, she can modify the contents of a collection. For instance, the programmer can modify the content of the first slot in a vector or change the value of a member of an instance of a class.
In some languages, even strings are mutable: their contents can be modified by a program at run time. For example, in Ruby, strings are mutable and in Java, StringBuffers are mutable.

In Clojure, by default, it is impossible for a program to modify the content of a slot in a vector, or any other collection. Likewise, it is impossible for a program to modify the content of a string. Data change in Clojure works in a similar way as a code repository like git: we never change the data itself, we create a new version of it. We will explore in details in Unit 7, how a program can change a data collection without modifying it.

Immutability is not an innovation of Clojure. It has been recommended in Software programming from the beginning. Clojure is the first pragmatic language to implement immutable data collections in an efficient way from a performance standpoint. (Before that, the only way to achieve immutability was through cloning. No need to say that it had performance issues.)

[KEY POINT] In Clojure, we never change the data itself, we create a new version of it and it is efficient from a performance standpoint.

[Extension] After the introduction of immutable data collections in Clojure, they have been implemented in many other languages like Ruby (https://github.com/hamstergem/hamster), Java (https://pcollections.org/) and Javascript (https://facebook.github.io/immutable-js). In Scala, they are part of the language since Scala 2.8.

Clojure remains unique in the sense that the data collections are natively immutable and that the whole language is designed around immutability.

Later in the book (in Unit 12), we will see that Clojure provides convenient ways for a program to manage state without compromising the value of immutability.

2.6 A powerful interactive development environment

A REPL (standing for Read-Eval-Print Loop) is a programming environment which enables the programmer to interact with a running Clojure program and modify it, by evaluating one code expression at a time.

The Clojure REPL gives the programmer an interactive development experience. When developing new functionality, it enables her to build programs first by performing small tasks manually, as if she were the computer, then gradually make them more and more automated, until the desired functionality is fully programmed. When debugging, the REPL makes the execution of her programs feel tangible: it enables the programmer to rapidly reproduce the problem, observe its symptoms closely, then improvise experiments to rapidly narrow down the cause of the bug and iterate towards a fix.

The REPL is very useful when you need to explore the API of a third party library.
Most Clojure IDEs integrate well with the REPL allowing the developer to evaluate pieces of code without leaving their editor.

Nowadays, most programming languages offer a REPL to the developer, sometimes it is called a console. But the Clojure REPL is more powerful as Clojure was designed with interactive development in mind.

2.7 JVM ecosystem

Like Scala, Kotlin and Groovy, Clojure is a JVM (Java virtual machine) language. It is compiled to Java bytecode (the instruction set of the JVM).

As a consequence,

1. Clojure programs run as regular Java programs on the JVM leveraging the stability and the performance of the JVM.
2. Clojure programs can be packaged as JAR (Java archive) and therefore deployed and executed on any platform that supports Java. Unlike Ruby or Python, Clojure doesn't require special support from the host machine.
3. Clojure programs have access to the rich Java ecosystem and can integrate with any Java library out there
4. Clojure libraries can be packaged as JAR and can be used in a native Java program.

This is one of the reasons that allowed Clojure to be a pragmatic language from day 1:

1. Server-side Clojure programs could be deployed on any PaaS (platform as a service) in the cloud.
2. Clojure programs could use any database drivers, web servers or analytics libraries available in Java. Later on, the Clojure community built wrappers to many Java libraries making it even easier to be used in a Clojure program.

2.8 Extensibility

2.8.1 Powerful macro system

As a LISP dialect, Clojure features a powerful macro system. Macros allow the language author and the developers to extend the language as they wish.

1. Authors can create terms to make the language be more concise and elegant without having to modify the compiler of the language.
2. Developers can add new terms to the language to meet their specific domain requirements.

Macros are also available in low-level languages like C but there macros are very complicated to write and limited because the syntax for macros is completely different that the syntax of other parts of the code.

In LISP, macros are much more powerful because of the homoiconicity (see section 2.4) : the syntax of the code is the same as the syntax of the data. Macros manipulate code in the same way functions manipulate data. As a consequence, all the data manipulation facilities are accessible by the macros.

2.8.2 Macros provided by Clojure

It is really mind blowing to notice that most of the terms defined by Clojure are not defined at the level of the compiler. They are defined at the level of `clojure.core`, the Clojure standard library.

The set of inner terms of Clojure, the terms that the Clojure compiler is aware of, is really small: it is made of 20 terms, called special forms. Most terms of the Clojure language are defined inside `clojure.core`; they are either functions or macros.

`clojure.core` provides around 70 macros:

- Some of them are syntactic sugar that allow the developers to write concise code. For instance, `when`, `cond`, `if-not`, `when-not`.
- Some of them are really at the core of the language. For instance, the form used for function definition (`defn`), the logical operators (`and`, `or`), the protocol definitions (`defprotocol`)

The fact that the inner parts of the language is made of a small number of terms is really valuable. It allows the Clojure compiler to remain small and simple even after more than 10 years of existence.

2.8.3 Custom macros

Developers in Clojure can use the macro mechanism in the same way it is used by `clojure.core`.

For instance, in Clojure we use the `defn` symbol to define a function. If you decide that for your project, you prefer to use a longer word like `def-function` or a shorter word like `f`, you can do that by creating your own macro.

And once you have your own symbol for defining functions, you can customize its functionality. For instance, you can decide that in your project, when a function is called it will add a line to a logger with the name of the function and the values of its arguments. No problem, you can do that with macros.

There is almost no limit to the extensibility of Clojure.

2.9 Elegance

As Harold Abelson and Gerald Jay Sussman wrote in their classic SICP (Structure and Interpretation of Computer Programs) book:

> *"programs must be written for people to read, and only incidentally for machines to execute"*

The elegance of the code we write matters. The main purpose of our code is to make the machine execute what we want. But we also need to make sure that other people (including us at some later time) are able to understand clearly our code.

Clojure features many aspects that make the code look elegant. Elegance is a matter of taste and of course, in order to appreciate this elegance you need first to get used to the syntax. Here are a couple of elements of Clojure that make it an elegant language, in my opinion:

1. No commas are required to separate the arguments in a function call or the elements in a collection
2. No return statement is needed at the end of a function body. Actually there is no return statement in Clojure
3. `clojure.core` provides several macros that allow the code to be concise and easy to read: `when`, threading macros `(->, ->>` and `as->)`
4. Clojure provides destructuring facilities to implicitly allocate local variables that contain values of some elements of a collection
5. In Clojure, everything is an expression. As a consequence, we can assign the result of an `if` expression into a variable.

2.10 Portability: Clojure and Clojurescript

Clojure is a hosted language that can be transpiled both to Java ByteCode and to Javascript. As so, a Clojure program can run on any Javascript environment:

1. Inside a web browser
2. Inside a browser extensions
3. As a node.js application
4. As a react native application
5. As a desktop application on top of Javascript frameworks like Electron

There are some differences between the Clojure and Clojurescript syntax and not all parts of Clojure are available in Clojurescript. And obviously the JVM ecosystem is not accessible from ClojureScript. But many Clojure libraries are ported to ClojureScript. On one hand, this book will focus on Clojure; on the other hand, once you are used to Clojure, it doesn't take too much effort to learn Clojurescript

Clojurescript allow us to leverage the key features of Clojure to any place where Javascript is available.

As the Clojure community says: "Clojure rocks. Javascript reaches".

Summary

In this Lesson we saw briefly at a high level what are the key features of the Clojure language. We can summarize Clojure key feature by writing that Clojure Clojure is a pragmatic, data-driven language that offers high power of expression to its developers. Most of those key features will be explored in details throughout the book.

Key Feature	Axis	Short Description	Value for developers	Book reference

Dynamically typed	Data Driven	Data types are determined at run time	Concise code	Unit 3
Rich collection types	Data Driven	Rich set of functions to manipulate data	Convenient data manipulation	Units 7, 14
Functional Programming	Data Driven	Functions are first class citizens	Simple code Power of expression Convenient data manipulation	Units 8, 9, 10
Immutability by default	Data Driven	Data never changes	Clear data flow Less bugs	Units 7, 12
Homoiconicity	Power of expression	Structure of the code is the same as the structure of the data	Uniform syntax Language extensibility IDE friendly	The whole book
Extensibility	Power of expression	Developers can extend the language	Ability to create custom syntax	Beyond the scope of the book
Elegance	Power of expression	Code is pleasant to read, usually concise	Code readability	The whole book
A powerful interactive development environment	Pragmatic	Ability to evaluate pieces of code in a standalone program	Debugging facilities Productive development workflow	Unit 2
JVM ecosystem	Pragmatic	Access to the JVM robust infrastructure and the multitude of Java libraries	High quality infrastructure Rich ecosystem Deployment facilities	Units 11
Testability	Pragmatic	Test framework is part of the language	Code quality	Beyond the scope of the book
Portability	Pragmatic	Targets JVM and Javascript (server side and client side)	Backend and frontend development	Beyond the scope of the book

Table 2.3 List of Clojure Key features

Clojure key features grouped by axis with a short description and the value for the developer that each feature provides and where in the book the feature is explored in greater details.

The key features exposed in this lesson are what allowed Clojure to be well received in the programmers community around the world. In the next lesson, we will see in what kind of projects the adoption of Clojure makes the most sense.

Lesson 3

Clojure in the industry

Takeaways

After reading this lesson, you will be able to:

- Describe what kind of projects are suited for Clojure and why they are suited
- Describe what kind of projects are not suited for Clojure and why they are not suited

Clojure is considered a general purpose language. In theory, it could be used for writing software in the widest variety of application domains. But in practice, there are some domains where the key features of Clojure make it a good choice and some domains where it is less appropriate to use Clojure.

In general, we can say that Clojure has been really well adopted in the field of web development.

3.1 What kind of projects are suited for Clojure?

Web Servers

A web server is a software program that uses the HTTP protocol to serve content to the World Wide Web.

There are many web servers frameworks available for Clojure. All of them run on top of Java web server frameworks and expose a Clojure interface to the programmer.

Writing a web server in Clojure allows you to get the best of two worlds:

1. In terms of the infrastructure, you rely on the quality of the Java web servers
2. In terms of the application, you enjoy all the benefits of the Clojure language

Clojure web servers can even be more scalable that Java web servers because Clojure has its own concurrency system provided by the highly scalable `core.async` library: `core.async` processes are more lightweight than Java threads therefore on the same machine, the number of concurrent processes is much higher that the number of Java threads.

Microservices

A Microservice is a software development technique that structures an application as a collection of loosely coupled services often called microservices.

Well-written microservices are highly scalable small, robust, well-tested and easy to deploy.

Typical microservices do data manipulation and access system resources like databases, message queues, external and internal APIs.

Writing a microservice in Clojure is really effective because:

1. Very often inside a microservice we need to manipulate data
2. Microservices architecture makes the most sense when the code base of each microservice remains small. Clojure concise syntax is really valuable in this context.
3. The powerful Clojure REPL is really useful when it comes to microservices. That's because since a lot of things rely on calling remote APIs often owned by other teams, being able to quickly experiment with them is really helpful in order to learn how they work and quickly be able to integrate with another microservice.

Therefore, it makes lot of sense to write microservices in Clojure.

Big Data

When building a big data system, data manipulation is at the core.

Therefore Clojure is a natural choice.

There are Java/Spark Big Data frameworks that you can use in Clojure and there are also Big data frameworks written from the ground up in Clojure.

Complex frontend web applications

The native language for frontend web application is Javascript.

But writing a complex program in Javascript is quite challenging as Javascript was designed at the beginning for writing document based web pages and not complex applications.

Over the years, Javascript has evolved both in terms of the language itself and in the ecosystem maturity and the build tools that it offers.

There are many languages out there that are transpiled to Javascript but Clojure stands out of them mainly because it embraces data immutability and nowadays most frontend frameworks like `react.js` encourage data immutability.

3.2 What kind of projects are not suited for Clojure?

Shell Scripts

For shell scripts to be executed inside a terminal, Clojure is not a perfect fit. Clojure startup time is a bit higher than native scripts or other dynamic languages like Ruby or Python. Also, the syntax for interacting with the shell (like running another script or accessing the environment variables) is not very straightforward.

Also, Clojure startup time is a bit high mostly due to the JVM startup time. In the context of a shell script it becomes really painful.

Usually, when writing a shell script you don't care too much about elegance and immutability. Your main purpose is to be able to get things done very quickly.

Infrastructure projects

Clojure is hosted on the JVM. In other words, Clojure is not a native language. As so, it is not appropriate for writing infrastructure projects where a close connection to the Operating system is required to make the system highly performant.

I wouldn't write a database or a load balancer in Clojure.

Mobile apps

Theoretically, you could write a Java mobile app in Clojure or a react native mobile app in ClojureScript. But the tooling required for that are not mature enough yet. Therefore it is currently quite cumbersome and require lot of manual interventions to make it happen.

Simple frontend web applications

The improvements of the Javascript language and the Javascript build tools like Webpack make it unnecessary to pay the price of writing non-native code for a simple frontend web application.

3.3 Summary

In this Lesson, we have seen in what kind of projects we can leverage the Clojure key features exposed in the previous lesson.

In general, Clojure adoption makes the most sense in the domain of web development mostly in the backend but also in the frontend.

Project	Best suited for Clojure	Main reasons
Web servers	Yes	Rely on solid Java frameworks Highly scalable
Microservices	Yes	Data manipulation facilities

		Concise syntax REPL
Big data	Yes	Data manipulation facilities
Complex frontend web applications	Yes	Immutability Solid software engineering framework
Shell Scripts	No	JVM startup time Cumbersome access to shell environment
Infrastructure projects	No	Non native
Mobile apps	No	No mature ecosystem
Simple frontend web applications	No	Non native

Table 3.1
This table summarizes the kind of projects where Clojure is best suited or not and the main reasons

1

Title template

1.1 Head 1

1.2.1 Head 2

HEAD 3 (heading 3)

Listing 1.1 (heading 4) .

```
code    <property name="src" location="src"/>   #A
(normal)
```

Normal

Table 1. (heading 5)

Typesetter notes (normal) Note: We don't print these

Figure 1. Captions (heading 6)

Sidebar head (heading 3) .

Sidebar (normal)