Unit 1

Get familiar with Clojure syntax

On one hand, Clojure syntax is so simple that it can be learnt in a couple of pages. On the other hand, it differs so much from other programming languages syntax that it will take time for your mind to get used to it.

The main purpose of this Unit is to get your mind used to the Clojure syntax until it starts to feel natural to you. We will expose the details of Clojure syntax for operations that you are already used to from other programming language only involving numbers and booleans.

The concepts are introduced slowly and different aspects of the syntax are exposed several times in a slightly different manner through the **Unit** without working in a real Clojure environment. The Clojure environment will be set up in **Unit 2**. We believe that it is more productive to train your mind with pencil and paper only before rushing on a keyboard.

After completing this **Unit**, you will be familiar with Clojure syntax where everything is wrapped with parentheses and without commas.

As we mentioned it in **Unit 0**, Clojure syntax is the same across the board. Once you get used to the syntax for manipulating numbers, you are ready to move forward and learn other parts of the language without having to learn any new syntax.

Units 1, 2 and 3 are oriented toward a noble purpose: they teach you how to write and test a real function in a Clojure environment on your machine.

Here are the high level concepts that will be explored in Units 1, 2 and 3:

- Unit 1: Clojure syntax, arithmetic and logical operations
- Unit 2: Clojure installation and evaluation of code in the REPL
- Unit 3: Variable definition, conditional branching and function definition

At the end of **Unit 3**, in the Capstone project, you will write a Clojure function that manipulates numbers and involves conditional branching. You will develop the code for this function and check that it works as expected inside a Clojure REPL installed on your machine.

In the current Unit, we will discuss the following topics:

- Lesson 4 explores how to write arithmetic expressions
- Lesson 5 explores how to compare numbers
- Lesson 6 explores how to write logic operations
- Lesson 7 discusses the structure of complex nested expressions

In this unit, we will compare Clojure syntax with the syntax of a language that you already know. This language could be Java, C#, Python, Ruby or Javascript. In the context of this book any of those languages is called a **Java-like language**.

This is a free excerpt of the book. In order to read the full book, please visit the book website.

Lesson 4

Arithmetic expressions

In order to get familiar with the Clojure syntax, we will learn how to deal with numbers in Clojure.

In the current **Lesson**, we will explore how to write arithmetic operations in Clojure. By arithmetic operations, we mean addition, subtraction, multiplication and division of numbers. We will discover how the Clojure syntax for arithmetic operations differs from the usual mathematical syntax. This will allow your mind to start getting used to the fact that in Clojure every compound expression is wrapped with parenthesis.

After completing this lesson, you will be able to:

- Understand simple arithmetic expressions in Clojure
- Write simple arithmetic expressions in Clojure
- Understand nested arithmetic expressions in Clojure
- Write nested arithmetic expressions in Clojure

Expressions and forms

In many programming languages, the language is made of functions, operators and statements. In Clojure, we have only expressions: every part of the language is an expression.

There are two kinds of expressions:

- **Data expressions:**
    - Primitive data expressions like numbers, booleans, strings...
    - Compound data expressions like arrays, hash maps...
- **Code expressions** like function calls, `if` expressions, variable definitions...

Code expressions are always wrapped with rounded parenthesis. Code expressions are also called *forms*. The name of the form is given by the first element of the expression. For instance, `(+ 1 2)` is a "+ form" and `(def my-variable 10)` is a "`def` form".

We multiply numbers with the "`*` form", we check for equality with the "= form", we define variables with the "`def` form", we write conditional flows with the "`if` form", we define functions with the "`defn` form" etc...

[EDITORIAL NOTE] Key point: The uniqueness of Clojure is that although each form has its own behaviour, the syntax of all the forms is the same. x

In Clojure, there are neither keywords nor reserved words, only forms: even terms like def or if that are at the core of the language are valid variable names. It's a bit silly to define a variable named if but it is totally valid.

Expression	Kind	Example	Meaning
A number	Primitive data expression	42	The number 42
A boolean	Primitive data expression	`true`	The boolean `true`
An array	Compound data expression	`[2 5 8]`	An array of numbers 2, 5 and 8
A + form	Code expression	`(+ 2 4)`	The addition of 2 and 4
A def form	Code expression	`(def my-var 42)`	Definition of a variable named `my-var` with value 42

Table 4.1 Examples of data and code expressions.

Code expressions are also called forms. The name of the form is given by the first element of the expression.

[EDITORIAL NOTE] Extension: In Clojure, there are three kind of forms: functions, macros and special forms. It is beyond the scope of this book to get into the differences between functions, macros and special forms.

Operations with two numbers

The syntax for arithmetic operations in most programming languages is the syntax that we use in our day to day life: `2 + 3, 4 * 7` etc...

Most programming languages use operators for arithmetic operations. The syntax of the arithmetic operators it differs from the syntax of regular function calls: the operators is placed between the two arguments while the function symbol is placed before its arguments like for example in `foo(1, 2)`.

Clojure takes a different approach: the syntax for arithmetic operations is exactly the same as the syntax for function calls. In fact, this is the same syntax for all the parts of the language: this is what we call a ***form*** or an ***expression***. We use both terms interchangeably.

In this lesson, we will focus on the simplest forms of the language: the arithmetic operations.

There are two main differences between the syntax of the forms that deal with arithmetic operations in Clojure and in the syntax of operators in other programming languages:

1. The operation symbol is placed before its arguments
2. The operation symbol and its arguments are wrapped in parentheses

The usual `2 + 3` arithmetic operation is written in Clojure: `(+ 2 3)`.

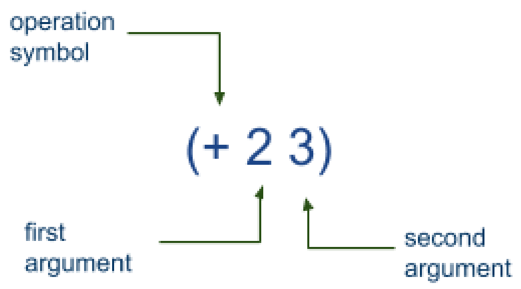


Figure 4.1
The structure of an addition with the + form illustrated by an example corresponding to the addition of 2 and 3

It may seem a bit unnatural for you at the moment, but the rules for arithmetic operations are quite simple:

1. Surround the operation with opening and closing parentheses
2. Inside the parentheses, place the operation symbol first - either `+`, `-`, `*` or `/` - followed by the operand(s)
3. Separate the elements within the parentheses by one or more whitespace characters

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.

[EDITORIAL NOTE] Keypoint: The exact same syntactic rules apply to all parts of the Clojure language.

Let's practice a bit by writing Clojure expressions for each one of the arithmetic operations.
We will start with expressions that involve only two numbers.

Let's write a Clojure expression for adding two numbers 5 and 7. In mathematical notation, it would be `5 + 7`.

In Clojure, we follow the following steps:

- Open and close the parenthesis: `()`
- First element inside the parentheses is the + symbol: `(+)`
- Second element inside the parenthesis is the first operand: the number 5: `(+ 5)`

- Third element inside the parenthesis is the second operand: the number 7: `(+ 5 7)`

The usual `5 + 7` becomes `(+ 5 7)` in Clojure.

As you might expect, the syntax for subtraction, multiplication and division is the same as the syntax for addition:

`5 - 7` becomes `(- 5 7)`

`5 * 7` becomes `(* 5 7)`

`5 / 7` becomes `(/ 5 7)`

[Editorial Discussion]: When we provide integers to the / form, we create a fraction. If we want a float number, we have to convert the fraction with the `float` form, like this: `(float (/ 5 7))`.

Remainder after division (modulus)

The modulus of two numbers `a` and `b` is the remainder after division of `a` by `b`.

In most programming languages, the modulus operation is done with the `%` operator: for instance, to calculate the remainder after division of 11 by 3, we write: `11 % 3`. In Clojure, we use the mod function and we write: `(mod 11 3)`. The result is 2 because 11 divided by 3 is 3 and the remainder is 2: `11 = 3*3 + 2`.

One common use case for mod is to check whether a number is odd or even by calculating the modulus by 2.
When `a` is even, `(mod a 2)` is equal `0`.
When `a` is odd, `(mod a 2)` is equal 1.

However, in Clojure we usually check whether a number is odd or even using `odd?` and `even?` functions that will be explored in Lesson 5.

Arithmetic Operations with more than two numbers

How do you write in mathematical notation an addition that involves 3 numbers like adding 2, 3 and 4?

`2 + 3 + 4`

You are probably so used to it that you don't notice any more that the + symbol is duplicated in this expression.

> **Key point:**: In Clojure, like we saw in the previous section, the operation symbol comes before the arguments. One advantage of having the operation symbol before the arguments is that when we want to operate on more than two numbers we don't need to duplicate the operation symbol.

In Clojure, adding 2, 3 and 4 is written like this: `(+ 2 3 4)`

We can add 4, 5 or any number of numbers using a single + symbol:

`(+ 7 2 8 9)` instead of `7 + 2 + 8 + 9`.

`(+ 7 2 8 9 11 12 19)` instead of `7 + 2 + 8 + 9 + 11 + 12 +19`.

Clojure developers enjoy the same freedom with all the arithmetic operations:

`(* 6 2 8 5)` instead of `6 * 2 * 8 * 5`.

For the subtraction and the division, we have to be careful with the rules of precedence

`(- 11 2 3)` is the same as `11 - 2 - 3`. We start from left to right: `(11 - 2) - 3` which equals 6.

`(/ 100 2 5)` instead of `100 / 2 / 5`. Here again, we start from left to right: `(100 / 2) / 5` which equals 10.

Rule	**Clojure**	**Math**
Operator position	before the operands	between the operands
Parenthesis	Required	Not Required
Separator	Whitespace	Whitespace
Number of operands	2 or more	Exactly 2

Table 4.1
A comparison of the syntactic rules for arithmetic operations in Clojure versus the usual Math notation.

Table 4.1 recaps the rules that we have seen so far.

[EDITORIAL NOTE]Discussion: There is a famous Zen Kōan (statement to meditate upon) that goes like this: "Two hands clap and there is a sound, what is the sound of one hand?". In Clojure, one can ask herself: "What happens when we add one number? When we multiply one number? When we subtract one number? When we divide one number?
Unlike the Zen Kōan, there is a concrete answer to each of those questions.

`(+ 3)` is 3 - An implicit `0` first argument is added
`(* 4)` is 4 - An implicit 1 first argument is added
`(- 3)` is -3 - An implicit `0` first argument is added
`(/ 2)` is 1/2 - An implicit 1 first argument is added

For the addition and the multiplication, one can even pass no argument:
`(+)` is `0`
`(*)` is 1

Now that we have seen how to handle arithmetic expression with numbers, let's move forward and see how to write arithmetic expressions where the arguments are themselves arithmetic expressions.

Nested Arithmetic Operations

The real fun begins when the arguments of the arithmetic expressions are themselves arithmetic expressions. The syntactic rule for arithmetic expressions is in fact recursive. In this section, we will decompose step by step each part of some nested expressions. We are going to slow down the pace, in order to make sure your brain becomes familiar with Clojure syntax. This is critical in order to be able to learn the next lesson. If you feel that you are already familiar at this point with the Clojure syntax, feel free to go over this section very quickly.

***[EDITORIAL NOTE]*Key point**: An operand of an arithmetic expression can be either a number or an arithmetic expression.

[EDITORIAL]Definition: nested expression
When one of the argument of an expression is itself an expression, we say that we have in hand a **nested expression**.

Let’s take a look at a nested arithmetic expression:

```
(+ (* 3 4) (* 6 7))
```

Can you guess what is the meaning of this nested arithmetic expression?
Let’s decompose the elements that occur at the first level in this nested expression:

1. The first element is the + symbol. Therefore it is an addition form
2. The first operand of the addition is itself an arithmetic expression `(* 3 4)`
3. The second operand of the addition is itself an arithmetic expression `(* 6 7)`

The expressions `(* 3 4)` and `(* 6 7)` are simple multiplications like the ones we saw in the previous section. We can decide to either decompose them explicitly or to try to grasp their meaning at once.

When we decompose also the elements that belong to the second level of nesting, we get:

1. The first element is the + symbol. Therefore it is an addition form
2. The first operand of the addition is a subexpression `(* 3 4)`
    a. The first element of this subexpression is the `*` symbol. Therefore it is a multiplication form
    b. The first operand of the multiplication is 3
    c. The second operand of the multiplication is 4
3. The second operand of the addition is a subexpression `(* 6 7)`
    a. The first element of this subexpression is the `*` symbol. Therefore it is a multiplication form
    b. The first operand of the multiplication is 6
    c. The second operand of the multiplication is 7

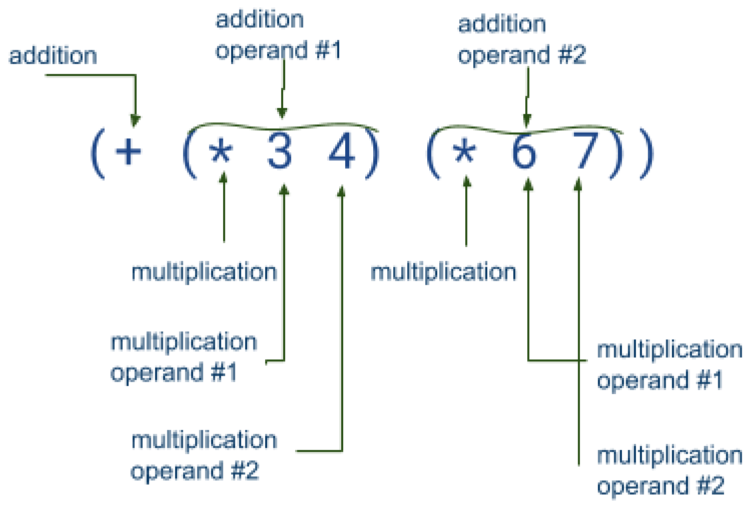

addition
addition operand #1
addition operand #2
(+ (* 3 4) (* 6 7))
multiplication
multiplication
multiplication operand #1
multiplication operand #1
multiplication operand #2
multiplication operand #2

Figure 4.2
The structure of a nested arithmetic operation illustrated by an example that involves the `*` form and the + form.

Either way, this expression is the addition of `3 * 4` and `6 * 7`. In math, it is written as: `(3 * 4) + (6 * 7)`.

(The result is 54, but who cares?)

Let’s take a look at a similar nested arithmetic expression where we reverse + and *.

```
(* (+ 3 4) (+ 6 7))
```

Let’s decompose the elements that occur at the first level of nesting in this expression:

1. The first element is the `*` symbol. Therefore it is a multiplication form
2. The first operand of the addition is itself an arithmetic expression `(+ 3 4)`

3. The second operand of the addition is itself an arithmetic expression `(+ 6 7)`

This expression is the multiplication of `3 + 4` and `6 + 7`. In math, it is written as: `(3 + 4) * (6 + 7)`.

(The result is 91, but who cares?)

If you really want, you can decompose also the elements that belong to the second level of nesting:

1. The first element is the `*` symbol. Therefore it is an addition form
2. The first operand of the addition is a subexpression `(+ 3 4)`
    a. The first element of this subexpression is the + symbol. Therefore it is an addition form
    b. The first operand of the additionis 3
    c. The second operand of the addition is 4
3. The second operand of the addition is a subexpression `(+ 6 7)`
    a. The first element of this subexpression is the + symbol. Therefore it is an addition form
    b. The first operand of the addition is 6
    c. The second operand of the addition is 7

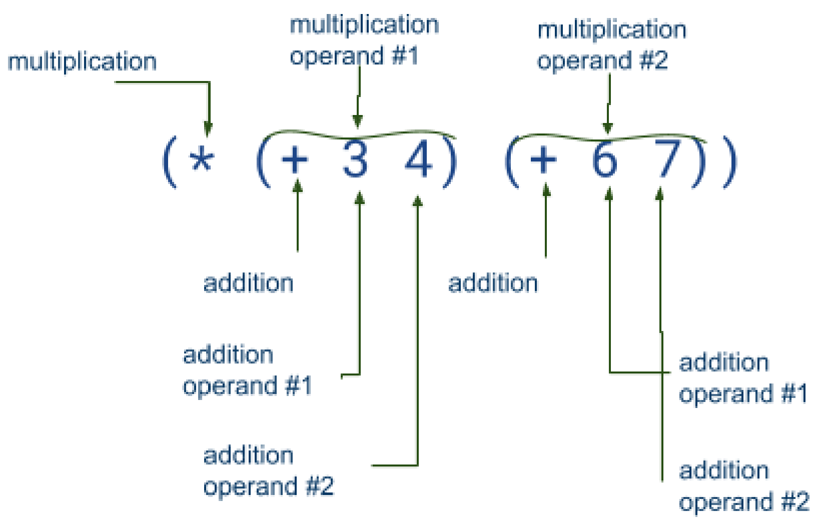


Figure 4.3
The structure of a nested arithmetic operation illustrated by an example that involves the `*` form and the `+` form.

Clojure	**Math**
`(* (+ 3 4) (+ 6 7))`	`(3 + 4) * (6 + 7)`
`(+ (* 2 5) (* 3 9))`	`(2 * 5) + (3 * 9)`
`(+ 4 (* 4 3))`	`4 + (4 * 3)`

Table 4.2
Some examples of nested arithmetic expressions in Clojure and the equivalent expression in Math notation.

Do you remember that in algebra, there are precedence rules? For instance, the multiplication takes precedence over the addition? It means that `(3 * 4) + (6 * 7)` can also be written without the parentheses like this: `3 * 4 + 6 * 7`. Imagine you write this expression inside a program. As a writer, you might think: "Hey that's cool, I was able to make my code shorter by removing the non required parentheses!"

But as a reader (it might be another developer or even yourself after several weeks), you impose on your brain an extra-step to figure out how to properly combine the parts of this expression.

In Clojure, parentheses are always required. There is no way to shorten this expression `(+ (* 3 4) (* 6 7))`. At first, it seems like it is cumbersome, but in fact it is valuable because it means the code is brain-friendly for the readers.

[EDITORIAL] Key point: Clojure syntax is concise and precise. It means that the code is brain friendly for readers.

Let’s recap the rules that we have seen so far:

Rule	**Clojure**	**Math**
Operator position	before the operands	between the operands
Parenthesis in simple expression	Required	Not Required
Separator	One or more whitespaces	One or more whitespaces
Number of operands	2 or more	Exactly 2
Parenthesis in nested expression	Required	Sometimes required, sometimes not required
Order of precedence	Not relevant	multiplication over addition etc...

Table 4.3
A comparison of the syntactic rules for arithmetic operations in Clojure versus the usual Math notation.

Summary

In this Lesson, we have started to explore the Clojure syntax in which every expression in wrapped with parenthesis. We have focused on arithmetic forms and compare the Clojure syntax with the usual mathematical syntax. The main differences are that in Clojure, the operations symbol comes before the numbers and that the parenthesis are always necessary. One advantage of the Clojure syntax over the usual syntax is that we can add or multiply more than two numbers without having to repeat the operation symbol. For instance, instead of writing `2 + 3 + 4`, with a duplicated + symbol, we write `(+ 2 3 4)` with a single + symbol.

In the next Lesson, we will continue our exploration of the syntax for number manipulation by focusing on number comparison forms. The syntax of arithmetic comparison forms in the next Lesson will be the same as the syntax of arithmetic operation forms of this Lesson: the comparison symbol will come before the numbers and the form will be wrapped in rounded parenthesis..

Try this

Ex 1:

Translate into mathematical notation the following Clojure expressions:

```
(+ 2 3)

(* 1 2)

(/ 2 4)

(- 3 5)
```

Ex 2:

Translate into mathematical notation the following Clojure expressions:

```
(+ 2 3 4 5)

(* 1 2 1 8)

(/ 2 4 3)

(- 3 5 9 8)
```

Ex 3:

Translate into Clojure expressions the following mathematical operations:

```
3 + 4

2 * 5

1 / 3

4 - 8
```

Ex 4:

Translate into Clojure expressions the following mathematical operations:

```
3 + 4 + 8 + 9

2 * 5 * 3 * 4

1 / 3 / 4

4 - 8 - 7 - 7
```

Ex 5:

Translate into Clojure expressions the following mathematical operations:

```
3 + (4 * 5)

(2 * 5) + (2 * 7)

1 / (3 + 9)

4 - (8 + 9)
```

Challenge: `5 + (9 * (7 - 3))`

Ex 6:

Translate into mathematical notation the following Clojure expressions:

```
(* (+ 2 3) (+ 4 5))

(+ (* 11 2) (* 2 8))

(/ (+ 2 4) 3)
```

`(- (* 3 5) (+ 9 8))`

Challenge: `(+ (* 4 3) (+ 7 8) 11)`

Answers

Ex 1:

Translate into mathematical notation the following Clojure expressions:

```
(+ 2 3)
Answer: 2 + 3

(* 1 2)
Answer: 1 * 2

(/ 2 4)
Answer: 2 / 4

(- 3 5)
Answer: 3 - 5
```

Ex 2:

Translate into mathematical notation the following Clojure expressions:

```
(+ 2 3 4 5)
Answer: 2 + 3 + 4 + 5

(* 3 2 7 8)
Answer: 3 * 2 * 7 * 8

(/ 2 4 3)
Answer:  (2/4)/3

(- 3 5 9 8)
Answer: ((3 - 5) - 9) - 8)
```

Ex 3:

Translate into Clojure expressions the following mathematical operations:

```
3 + 4
Answer: (+ 3 4)

2 * 5
Answer: (* 2 5)

1 / 3
Answer: (/ 1 3)

4 - 8
Answer: (- 4 8)
```

Ex 4:

Translate into Clojure expressions the following mathematical operations:

```
3 + 4 + 8 + 9
Answer: (+ 3 4 8 9)

2 * 5 * 3 * 4
Answer: (* 2 5 3 4)

(1 / 3) / 4
Answer: (/ 1 3 4)

((12 - 8) - 3) - 7
Answer: (- 12 8 3 7)
```

Ex 5:

Translate into Clojure expressions the following mathematical operations:

```
3 + (4 * 5)
Answer: (+ 3 (* 4 5))

(2 * 5) + (2 * 7)
Answer: (+ (* 2 5) (* 2 7))

1 / (3 + 9)
Answer: (/ 1 (+ 3 9))

4 - (8 + 9)
Answer: (- 4 (+ 8 9))
```

Challenge: `5 + (9 * (7 - 3))`
```
Answer: (+ 5 (* 9 (- 7 3)))
```

Ex 6:

Translate into mathematical notation the following Clojure expressions:

```
(* (+ 2 3) (+ 4 5))
Answer: (2 + 3) * (4 + 5)

(+ (* 11 2) (* 2 8))
Answer: (11 * 2) + (2 * 8)

(/ (+ 2 4) 3)
Answer: (2 + 4) / 3

(- (* 3 5) (+ 9 8))
Answer: (3 * 5) - (9 + 8)
```

Challenge: `(+ (* 4 3) (/ 7 8) 11)`
```
Answer: (4 * 3) + (7 / 8) + 11
```

Lesson 5
Arithmetic comparisons

Arithmetic comparisons in Clojure follow the same syntax as arithmetic operations where the symbol comes before the numbers and the expression is wrapped with parenthesis.

After completing this lesson, you will be able to read and write Clojure expressions for:

- comparing numbers
- comparing arithmetic expressions
- checking whether a number is positive or negative
- checking whether an arithmetic expression is positive or negative
- calculating the maximum and the minimum of several numbers

Equality check

In many languages, the equal sign is used for two kind of operations:

- assignment of a value into a variable, usually with a single equal sign =
- equality check between values, usually with two equal signs ==

For instance in Python, variable assignment is done wit a single equal sign:

```
num = 1
```

And equality check is done via the == symbol:

```
2 == 3
```

(In javascript, we even have the === symbol, which behaves slightly differently from the == symbol.)

In Clojure, assignment and equality check are well separated by completely different symbols. The = symbol is only for quality check. Variable assignment is done with the `def` form that will be explored in Lesson 11.

Checking whether two numbers are equal is written like this:

```
(= 2 3)
```

The syntax for equality check is exactly the same as the syntax for arithmetic operations that we introduced in the previous lesson. The difference is that an expression like `(= 2 3)` returns a boolean, either `true` or `false`, while an expression like `(+ 2 3)` returns a number.

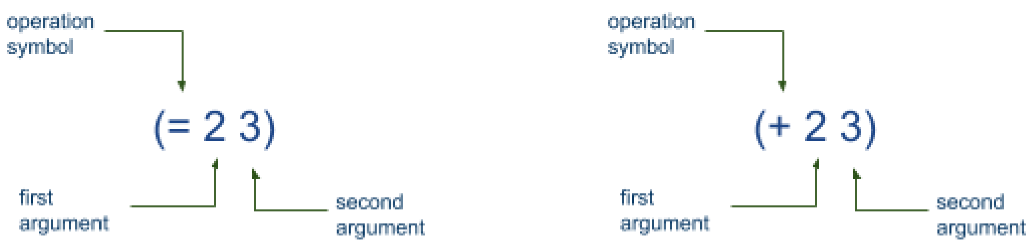


Figure 5.1
Similarities between the structure of the = form and the + form

Although the meaning of `(= 2 3)` and `(+ 2 3)` is completely different, their syntax is is exactly the same. This property is quite unique to Clojure (and other languages based on LISP): We will see through the book that all the parts of the language obey the same syntactic rules.

The fact that equality check follows the same syntax as arithmetic operations allows us, like with arithmetic operations, to compare more than two numbers, simply by adding more elements inside the parenthesis, just like this:

```
(= 2 3 4)
```

In other languages, equality check is done via an operator, where the equality sign is placed between the two numbers to be compared. The equivalent Java or C# expression would be:

```
2 == 3 && 3 == 4
```

So in order to compare 3 numbers, we need to make use of an additional operator: the && logical operator.

Please take a moment to compare the simplicity of the Clojure expression `(= 2 3 4)` with the complexity of the Java expression `2 == 3 && 3 == 4`. The root cause of this complexity is the fact that the == is a binary operator and not a function. Binary operators require the operator symbol to be placed between the two numbers to be compared. The syntactic rules for operators and functions are not the same. The syntax of a function call is more flexible than the syntax of a binary operator.

[EDITORIAL NOTE] Extension: In Python, we can compare three numbers without having to introduce the extra and operator. The reason is that the == operator can be chained. Therefore, we can write `2 == 3 == 4` and it returns the same result as `2 == 3 and 3 == 4`.

This is one of the advantages of the uniformity of the Clojure syntax. Once again, let's emphasize it, as it is essential to the understanding of Clojure syntax:

[EDITORIAL NOTE] Keypoint: In Clojure all parts of the language follow the same syntax: function calls, arithmetic operations, arithmetic comparisons, `if` expressions, variable definition etc...

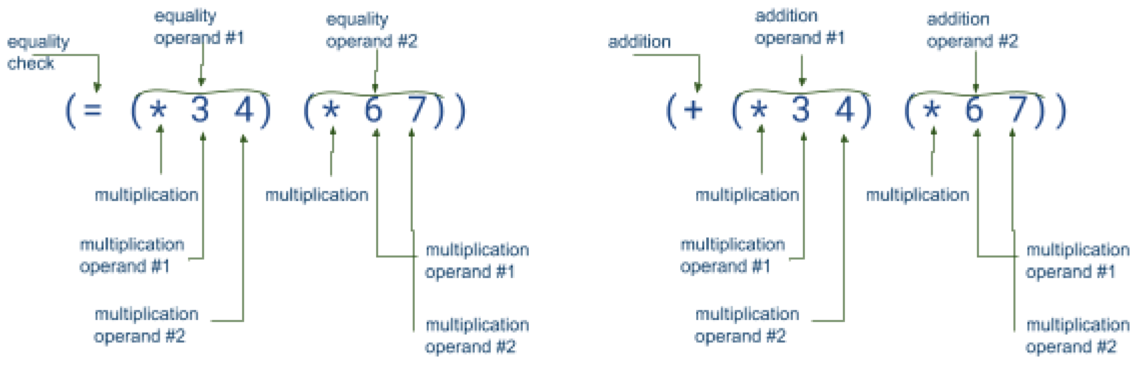
Let's see now how we compare the results of two arithmetic expressions (e.g. multiplication). For that purpose, we need to build a nested expression that contains a `*` form inside the `=` form:

```
(= (* 3 4) (* 6 7))
```

Take a moment to look at this expression and see if you can understand its meaning.

It compares the value of `(* 3 4)` with the value of `(* 6 7)`. In Python, this nested expression would be written as: `3 * 4 == 6 * 7`. The two values are not equal, therefore the value of this nested expression is `false`.

Here again, from a syntactic perspective, the nested comparison looks the same as the nested addition that we saw in previous lesson `(+ (* 3 4) (* 6 7))`



equality check
equality operand #1
equality operand #2
(= (* 3 4) (* 6 7))
multiplication
multiplication operand #1
multiplication operand #2
multiplication
multiplication operand #1
multiplication operand #2
addition
addition operand #1
addition operand #2
(+ (* 3 4) (* 6 7))
multiplication
multiplication operand #1
multiplication operand #2
multiplication
multiplication operand #1
multiplication operand #2

Figure 5.2
Similarities between the structure of a nested = form and a nested + form. The expression on the left corresponds to `3 * 4 == 6 * 7` and the expression on the right corresponds to `3 * 4 + 6 * 7`.

Let's conclude this section with a table that compares some Clojure = forms with their Java-like equivalent.

Clojure	**Java-like syntax**
`(= 2 3)`	`2 == 3`
`(= 2 3 4)`	`2 == 3 && 3 == 4`
`(= (* 3 4) (* 6 7))`	`3 * 4 == 6 * 7`

Table 5.1
Some examples of equality check in Clojure and the equivalent Java-like syntax. The `(= 2 3 4)` in the second row illustrates the advantage of the fact that in Clojure = is a function and not an operator. We can compare three numbers without having to introduced the && operator.

Arithmetic comparisons

In Clojure, we have the same arithmetic comparison capabilities as in other programming languages:

- = for equal
- > for greater than
- >= for greater or equal
- < for less than
- <= for less or equal

The big difference is that in Clojure the arithmetic comparison are functions and not operators. In fact, in Clojure, there are no operators.

All the arithmetic comparison functions follow the same syntax as any other expression of the language. Let's write the syntax rules in details one last time, to make sure your mind is fully acquainted with them:

1. Opening and closing parenthesis
2. First element inside the parentheses is the function symbol: `>, >=, <, <= or =`
3. Second element inside the parentheses is the first operand
4. Third element inside the parentheses is the second operand
5. The elements inside the parenthesis are separated by a whitespace

Checking whether a number is equal to another number is written like this:

```
(= 2 3)
```

Checking whether a number is greater than another number is written like this:

```
(> 2 3)
```

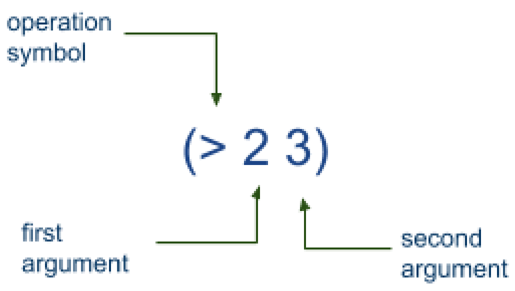


Figure 5.3
The structure of a arithmetic comparison with the > form illustrated by an example that is equivalent to 2 > 3.

The other arithmetic comparison forms work in the exact same way.

Checking whether a number is greater or equal to another number is written like this:

```
(>= 2 3)
```

Checking whether a number is smaller than another number is written like this:

```
(< 2 3)
```

Checking whether a number is smaller or equal to another number is written like this:

```
(<= 2 3)
```

[EDITORIAL: Keypoint] Pay attention to the order of arguments inside an arithmetic comparison form: the first argument is compared with the second number. For instance, (< 2 3) is equivalent to 2 < 3.

A special use case for the arithmetic comparison is when we want to check whether a number is comprised between two numbers. For instance, checking that a customer is a teenager by checking if the age of a customer is between 14 and 18. Or checking whether a product purchase occurred in the first quarter of the year by checking if the month of the purchase is between January and March.

In Java or C#, the code for checking whether a customer is a teenager would look like this:

```
14 <= age && age <= 18
```

It is quite cumbersome to introduce the && operator for such a simple comparison.
In Clojure, the code for checking whether a customer is a teenager would look like this:

```
(<= 14 age 18)
```

[EDITORIAL NOTE] Extension: In Python, we can compare three numbers without having to introduce the extra and operator. The reason is that the comparison operators can be chained. Therefore, we can write `14 <= age <= 18` and it returns the same result as `14 <= age and age <= 18`.

Now, let's see how we mix arithmetic operations and arithmetic comparisons in a single nested expression. For instance, let's write in Clojure the following python expression that checks whether 3*4 is greater than 6*7:

```
3 * 4 > 6 * 7
```

In Clojure, it becomes:

```
(> (* 3 4) (* 6 7))
```

An important thing to notice is that in Clojure, you cannot write arithmetic comparison without the parenthesis. It means that the operator precedence rules are not relevant in Clojure for the simple reason that in Clojure, there are no operators. That's good news for you because it means that in a Clojure job interview, you will never be asked whether the equality operator takes precedence over the && operator or any similar mind troubling questions. Another positive consequence of the absence of operators in Clojure is that when you read pieces of code written by other that involves arithmetic comparison, the meaning of the code is very clear and explicit.

[EDITORIAL]Keypoint: In Clojure, there are no operator precedence rules for the simple reason that in Clojure, there are no operators.

Let’s conclude this section by a table that compare some arithmetic expressions, Clojure vs Java-like languages:

Clojure	**Java-like languages**
`(> 2 3)`	`2 > 3`
`(= 2 4)`	`2 == 4`
`(<= 14 age 18)`	`14 <= age && age <= 18`
`(> (* 3 4) (* 6 7))`	`3 * 4 > 6 * 7`

Table 5.2
Comparison of some arithmetic comparison expressions: Clojure vs. Java-like languages. The `(<= 14 age 18)` in the second row illustrates the advantage of the fact that in Clojure arithmetic comparison is done with functions and not with operators. We can compare three numbers without having to introduced the && operator.

Max and Min

There is another aspect of number comparison: calculating the maximum and the minimum of several numbers. Clojure provides two functions `max` and `min` for this purpose. Both receive one or more numbers and they follow the same syntax of the arithmetic comparison forms.

Let's look at a `max` form first:

`(max 2 3 5)` calculates the maximum between `2`, `3` and `5` which is `5`.

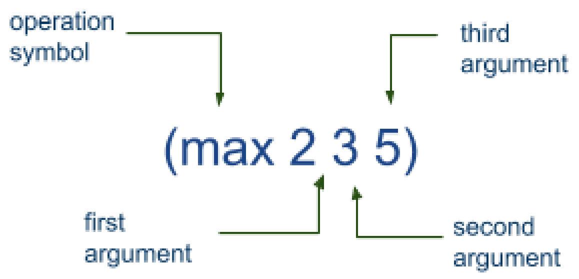


Figure 5.4
The structure of the `max` form illustrated by an example that calculate the maximum of three numbers.

`(min 2 3 5)` calculates the minimum between `2`, `3` and `5` which is `2`.

When you pass a single number to max or `min`, it returns the number itself:

`(max 42)` is 42.
`(min 42)` is 42.

Basic Number Property Functions

In Clojure, we have a couple of functions that check basic properties of numbers:

- pos? - check whether a number is strictly positive
- neg? - check whether a number is strictly negative
- `zero?` - check whether a number is zero
- odd? - check whether a number is odd (only valid for integers)
- even? - check whether a number is even (only valid for integers)

The syntax for all the above functions is the same as the syntax of the arithmetic operations we saw earlier except that the above functions accept only a single argument.

Checking whether a number is positive is written like this:

```
(pos? 2)
```

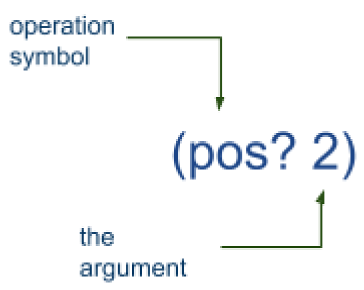


Figure 5.5
The structure of the pos? form illustrated by an example that checks whether 2 is a positive number.

Notice that ? is allowed as part of the name of a function. In Clojure (like in Ruby), there is a convention to name functions that return either `true` or `false` with an ending question mark.

- Checking whether a number is negative is written like this: `(neg? 2)`
- Checking whether a number is equal to zero is written like this: `(zero? 2)`
- Checking whether a number is odd is written like this: `(odd? 2)`
- Checking whether a number is even is written like this: `(even? 2)`

[EDITORIAL] Discussion: Why do we need this functions at all?
Checking whether a number x is equal to 0 could also easily be written without the `zero?` function, using the = form like this: `(= x 0)`. The difference is in the readability of the code. `(zero? x)` conveys more intention to the reader than `(= x 0)`. In terms of intention, checking whether x is equal to zero is not the same as checking whether a number is equal to 1. Zero is a special case. It deserves its own function. A similar argument applies to `pos?`, `neg?`, `odd?` and `even?`.

Like we saw earlier, we can pass to any of this functions an expression as argument.

For instance:

`(zero? (- 3 3))` checks whether `(- 3 3)` is zero, therefore it returns `true`.

or

`(pos? (* (- 3 4) (+ 2 4)))` checks whether `(* (- 3 4) (+ 2 4))` is positive, therefore it returns `false`.

[EDITORIAL NOTE]Discussion: What happens when we check whether a float number is odd?
The argument to odd? and even? must be an integer. If we pass a float number and try to execute something like (odd? 2.3) an exception will be thrown by the code of the odd? function at run time. We will discuss exceptions in Clojure later in the book.

[EDITORIAL NOTE]Discussion: What happens whe we pass more than one argument?
All the functions mentioned in this lesson accept only a single argument. If you try to pass more than 1 argument like for instance (pos? 2 3), it will throw an exception. The exception is thrown by Clojure itself before even reaching the code of pos?. We will discuss exceptions in Clojure later in the book.

Summary

In this **Lesson**, we have explored how to compare numbers in Clojure. The main difference between Clojure and other programming languages is that in Clojure there are no operators. All the arithmetic comparison are done with functions. On advantages of the Clojure syntax over the usual syntax for number comparison is that we can compare more than two numbers with a single operation. We have seen also that in Clojure, there is no precedence rules to remember because the parenthesis are mandatory. In addition to the 5 arithmetic comparison functions =, >, >=, < and <=, we have also discussed:

- `pos?` that checks whether a number is strictly positive
- `neg?` that checks whether a number is strictly negative
- `zero?` that checks whether a number is zero
- `odd?` that checks whether a number is odd (only valid for integers)
- `even?` that checks whether a number is even (only valid for integers)

So far, we have been manipulating only numbers. In the next **Lesson**, we will start to manipulate booleans and explore Clojure syntax for logic operations.

Try this

Ex 1:

Write the following Clojure expressions in Java notation:

```
(= 3 4)
```

```
(= 2.3 4.5)
```

```
(> 3 4)
```

```
(<= 5 6)
```

Ex 2:

Write the following Clojure expressions in Java notation:

```
(= 3 4 5)
```

```
(= 2 3 4 5 6)
```

`(< 3 age 9)` assuming that age is a number

```
(>= 4 5 0)
```

Ex 3:

Write the following Clojure expressions in Java notation:

```
(= 3 (+ 4 5))
```

```
(= (+ 2 3) (+ 4 5))
```

```
(> (* 2 3) (+ 4 5))
```

```
(< 8 (+ 4 5))
```

Ex 4:

What is the meaning of the following Clojure forms?

```
(zero? 3)
```

```
(pos? -5)
```

```
(neg? 19)
```

```
(odd? 9)
```

```
(even? 7)
```

Ex 5:

Why is there a question mark at the end of each of the functions of the following Clojure forms?

```
(zero? 3)
(pos? -5)
(neg? 19)
(odd? 9)
(even? 7)
```

Ex 6:

Convert the following Java expressions into Clojure expressions:

`3 == 4`

`2.2 == 9.8`

`4 > 6`

`5 <= 8`

Ex 7:

Convert the following Java expressions into Clojure expressions:

`3 == 4 && 4 == 5`

`3 == 4 && 4 == 5 && 5 == 6`

`10 > age && age > 3` assuming that age is a number

`5 <= 8 && 8 <= 2`

Ex 8:

Convert the following Java expressions into Clojure expressions:

`3 == 4 + 9`

`2 + 6 == 5 + 8`

`2 * 7 > 4 + 1`

`8 < 5 + 2`

Ex 9:

Write a Clojure expression that checks whether:

`3` is equal to zero

`-6` is positive

`4 - 3` is negative

`2 + 5` is odd

`14 + 9 + 3` is even

Answers

Ex 1:

Write the following Clojure expressions in Java notation:

```
(= 3 4)
Answer: 3 == 4

(= 2.3 4.5)
Answer: 2.3 == 4.5

(> 3 4)
Answer: 3 > 4

(<= 5 6)
Answer: 5 <= 6
```

Ex 2:

Write the following Clojure expressions in Java notation:

```
(= 3 4 5)
Answer: 3 == 4 && 4 == 5

(= 2 3 4 5 6)
Answer: 2 == 3 && 3 == 4 && 4 == 5 && 5 == 6
```

`(< 3 age 9)` assuming that age is a number
```
Answer: 3 < age && age < 9

(>= 4 5 0)
Answer: 4 >= 5 && 5 >= 0
```

Ex 3:

Write the following Clojure expressions in Java notation:

```
(= 3 (+ 4 5))
Answer: 3 == (4 + 5)

(= (+ 2 3) (+ 4 5))
Answer: (2 + 4) == (4 + 5)

(> (* 2 3) (+ 4 5))
Answer: (2 * 3) > (4 + 5)

(< 8 (+ 4 5))
Answer: 8 < (4 + 5)
```

Ex 4:

What is the meaning of the following Clojure forms?

`(zero? 3)`
Answer: it checks whether 3 is zero.

`(pos? -5)`
Answer: it checks whether -5 is a positive number.

`(neg? 19)`
Answer: it checks whether 19 is a negative number.

`(odd? 9)`
Answer: it checks whether 9 is an odd number.

`(even? 7)`
Answer: it checks whether 7 is an even number.

Ex 5:

Why is there a question mark at the end of each of the functions of the following Clojure forms?

```
(zero? 3)
(pos? -5)
(neg? 19)
(odd? 9)
(even? 7)
```

Answer: The reason for the ending question mark is that all of this functions return either `true` or `false`. In Clojure, It is a convention to have an ending question mark for functions that return either `true` or `false`.

Ex 6:

Convert the following Java expressions into Clojure expressions:

```
3 == 4
Answer: (= 3 4)

2.2 == 9.8
Answer: (= 2.2 9.8)

4 > 6
Answer: (> 4 6)


5 <= 8
Answer: (<= 5 8)
```

Ex 7:

Convert the following Java expressions into Clojure expressions:

```
3 == 4 && 4 == 5
Answer: (= 3 4 5)


3 == 4 && 4 == 5 && 5 == 6
Answer: (= 3 4 5 6)
```

`10 > age && age > 3` assuming that age is a number
`Answer: (> 10 age 3)`

```
5 <= 8 && 8 <= 2
Answer: (<= 5 8 2)
```

Ex 8:

Convert the following Java expressions into Clojure expressions:

```
3 == 4 + 9
Answer: (= 3 (+ 4 9))

2 + 6 == 5 + 8
Answer: (= (+ 2 6) (+ 5 8))


2 * 7 > 4 + 1
Answer: (> (* 2 7) (+ 4 1))
```

`8 < 5 + 2`
`Answer: (= 8 (+ 5 2))`

Ex 9:

Write a Clojure expression that checks whether:

`3` is equal to zero
`Answer: (zero? 3). We could also write (= 3 0). But (zero? 3) is more elegant.`

`-6` is positive
`Answer: (pos? -6). We could also write (> -6 0). But (pos? 3) is more elegant.`

`4 - 3` is negative
`Answer: (neg? (- 4 3)). We could also write (< (- 4 3) 0). But (neg? (- 4 3)) is more elegant.`

`2 + 5` is odd
`Answer: (odd? (+ 2 5)).`

`14 + 9 + 3` is even
`Answer: (even? (+ 14 9 3)).`

Lesson 6

Logic operations

There are three basic logic operations that a programmer manipulates on a daily basis: negation, disjunction and conjunction also named *logical not*, *logical or* and *logical and*.

In many languages, we have a symbol for each operation:

- `!` for logical not
- `&&` for logical and
- `||` for logical or

In Clojure, we use the operation name itself:

- `not` for logical not
- `and` for logical and
- `or` for logical or

The fact that we use the operation name itself makes the code more readable.

In this lesson, we will introduce the two boolean values: `true` and `false`, and get a first introductory meeting with the Clojure null value: `nil`.

The most natural arguments for logic operations are the booleans. But Clojure being a dynamically typed programming language, it doesn't limit the usage of logic operations to booleans. In this Lesson, we will learn how numbers and `nil` are handled by logic operations.

After completing this lesson, you will be able to:

- Understand Clojure expressions that deal with logic conditions involving one logic operation
- Write Clojure expressions that deal with logic conditions involving one logic operation
- Understand Clojure expressions that deal with logic conditions involving several logic operations
- Write Clojure expressions that deal with logic conditions involving several logic operations

Boolean values, nil and truthiness

Clojure, like other dynamic programming languages (python, ruby, javascript etc...), does not restrict the logic operations to boolean values. Therefore, Clojure has to define exactly what values behave like `false` (falsy values) and what values behave like `true` (truthy values). In Clojure, the rule is simple: `false` and `nil` are the only falsy values. All other values are truthy.

If you are familiar with the notions of truthy and falsy values from a language like python, ruby or javascript, you can read this section quickly. If you come from a statically type language like C# or Java, this section might be a bit harder to digest.

Like any other language, Clojure has two boolean values: `true` and `false`. However, every expression is either truthy or falsy.

[EDITORIAL NOTE]Key Point: In Clojure, like in python, ruby and javascript, every expression can be checked for truthiness.

The *meaning* of x being truthy or falsy is whether x is considered to be true or false by the logic operations: Indeed the logic operations handle truthy values exactly as they handle `true` and they handle falsy values exactly as they handle `false`.

Clojure also has a `nil` value that has many interesting properties. In the context of the current Lesson, the most important property of `nil` is that `nil` is treated like `false` by the logic operations: We say that nil is falsy.

You don't need to be worried too much by `nil` for the moment. We will explore `nil` in greater detail later in the book.

The rule for *deciding* whether x is truthy or falsy is very simple: the only falsy values are `nil` and `false`. All other values are truthy.

- All the numbers are truthy, including 0.
- All the strings are truthy, including the empty string.
- All the arrays are truthy, including the empty array.
- All the functions are truthy.
- Only `false` and `nil` are falsy.

In this Lesson, we are going to explore three logical forms: `and`, `not` and `or`.

Theses logical forms are not limited to boolean values and `nil`. They can deal with any type of values.

[EDITORIAL NOTE]Key Point: In Clojure, the only **falsy** values are `nil` and `false`. All other values are **truthy**.

Logical negation: not

The behaviour of `not` is quite simple: `not` returns `true` when its argument is falsy and `false` when its argument is truthy.

The syntax of `not` is the same as the syntax of all the expressions we have encountered so far except that it accepts only a single argument.

> `(not true)` is `false` because `true` is truthy.
> `(not false)` is `true` because `false` is falsy.
> `(not nil)` is `true` because `nil` is falsy.
> `(not 5)` is `false` because 5 is truthy.
> `(not 0)` is `false` because `0` is truthy.

The main use case for `not` is checking whether a condition holds or not. Let's look at some examples:

> `(not (= 2 2))` is `false` because `(= 2 2)` is `true`. Indeed, 2 equals 2.
> `(not (= 4 2))` is `true` because `(= 4 2)` is `false`. Indeed, 4 doesn't equal 2.
> `(not (pos? 2))` is `false` because `(pos? 2)` is `true`. Indeed, 2 is positive.
> `(not (neg? 2))` is `true` because `(neg?)` is `false`. Indeed, 2 is not negative.

The `not` form returns a boolean, therefore it is totally fine to compose an expression with a nested `not` function call, like this:

`(not (not false))`
or
`(not (not (= 4 2)))`

When `not` is passed a boolean, subsequent calls to `not` have a cancellation effect on each other: The value of an expression that calls `not` twice on a boolean value x is x. For instance, the value of `(not (not false)` is the same as the value of `false` and the value of `(not (not (= 4 2)))` is the same as the value of `(= 4 2)` which is `false`.

When we pass a number to a double not, for instance `(not (not 5))`, we don't get 5 but `true`. The reason is that not always returns either `true` or `false`. A double `not` returns `true` when the argument is truthy and `false` when the argument is falsy.

If you understand each row of the following table, you can claim that you understand the `not` form perfectly.

Expression	**Value**	**Explanation**
`(not (not false))`	`false`	`false` is falsy
`(not (not true))`	`true`	`true` is truthy
`(not (not 5))`	`true`	`5` is truthy
`(not (not 0))`	`true`	`0` is truthy
`(not (not nil))`	`false`	`nil` is falsy
`(not (not (> 4 2)))`	`true`	`(= 4 2)` is truthy
`(not (not (= 4 2)))`	`false`	`(= 4 2)` is falsy

Table 6.1 Cancellation effect of a double `not` expression.
When the argument is a boolean, the cancellation effect is accurate: (not (not x)) is the same as x. When the argument is not a boolean, the cancellation effect is not accurate: (not (not x)) is either `true` or `false` according to whether x is truthy or falsy.

Logical and

The and form allows us to check whether several values or expressions are truthy.

Basically the and form returns a truthy value when all its arguments are truthy and `false` when one of its arguments is falsy (`false` or `nil`). We are going to describe this behaviour more precisely throughout this section.

The syntax of the and form is simple but there are some nuances in its behaviour. Let's see the syntax first.

The syntax of the and form is exactly the same as the arithmetic comparison functions we saw earlier (`=`, `>` and `<`).

A call to and with two arguments looks like this: `(and true false)`

Like the arithmetic comparison functions, and can receive more that two arguments like for instance in this expression: `(and true false true false)`.

and can also receive one argument: `(and true)` is perfectly valid.

and can even receive zero argument: (and) is also valid. A bit silly but valid.

Now let's explore the behaviour of and. When all the arguments are boolean, the rule is simple: and returns `true` when all its arguments are `true` and `false` when one of its arguments is `false`.

`(and true)` is `true`.
`(and false)` is `false`.

`(and true false)` is `false` because one of the arguments is `false`.
`(and true true)` is `true` because all the arguments are `true`.

`(and true false true)` is `false` because one of the arguments is `false`.
`(and true true true)` is `true` because all the arguments are `true`.

`(and (> 5 3) (> 5 19))` is `false` because the second expression is `false`: 5 is not greater than 19.

`(and (> 5 3) (> 5 1))` is `true` because all the arguments are `true`: 5 is greater than 3 and 5 is also greater than `1`.
`(and (pos? 12) (neg? 4))` is `false` because the second expression is `false`: 4 is not negative.

Finally, let's take a look at the behaviour of and when it receives no arguments: `(and)` is a weird situation that falls under the case of ***"all the arguments are*** `true`***"*** in a special way because indeed none of the arguments are false. Therefore, `(and)` is `true`.

Like in other programming languages, and is smart enough not to evaluate arguments when it is not necessary. and goes over its arguments from left to right. As soon as it encounters an expression that is `false`, it stops the evaluation and returns `false`.
For instance, in the expression `(and (neg? 12) (neg? 4))`, `(neg? 12)` is false. Therefore `(neg? 4)` will not be evaluated at all and the whole expression will be evaluated to `false`.

non-boolean arguments

So far, we have seen how and behaves when all its arguments are boolean.

Like for `not`, the arguments of and are not required to be booleans. You can pass any value or expression to and.

and goes over its arguments from left to right:

- When one of the arguments is falsy (either `false` or `nil`), its value is immediately returned (without evaluating the remaining arguments).
- When all the arguments are truthy, the value of the last argument is returned.

`(and true false true)` is `false` because the first falsy arguments is `false`.

`(and true nil true)` is `nil` because the first falsy arguments is `nil`.

`(and true true 8)` is 8 because all the arguments are truthy and the last argument is 8.
`(and 4 5 9)` is 9 because all the arguments are truthy and the last argument is 8.

Case	**Behavior**	**Example**	**Result**
0 argument	return true	`(and)`	`true`
1 argument	return true if the argument is truthy	`(and true)`	`true`
2 arguments	return true if both arguments are truthy	`(and true false)`	`false`
any number of arguments	return true if all the arguments are truthy	`(and true true false true)`	`false`
Non-boolean values	return false if one arguments is falsy return the last argument when all the arguments are truthy	`(and 4 5)`	`5`

Table 6.2
The behaviour of the and form according to different types and numbers of arguments.
For each case, the behavior of and is explicitly stated and illustrated by a concrete example with the result of the evaluation of the expression in the example.

Let’s practice now on combining and with `not`.

`(not (and true false)) is true because (and true false) is false.`

`(and true (not false))` is `true` because all the arguments are `true`.

`(and (= 2 2) (not (= 4 5)))` is `true` because all the arguments are `true`.

`(and (not (= 2 3)) (not (= 4 4)))` is `false` because the last expression is `false`.

Let's complete our logical operations portfolio by exploring the syntax and the behaviour of `or`.

Logical or

`or` allows you to check whether there is a least one truthy expression in a series of expressions.

Basically `or` returns a truthy value when one of its arguments is truthy and `false` when all of its arguments are falsy (`false` or `nil`). We are going to describe this behaviour more precisely through this section.

The syntax of `or` is simple but there are some nuances in its behaviour. Let's see the syntax first.

The syntax of `or` is exactly the same as thes syntax of `and`.
A call to `or` with two arguments looks like this: `(or true false)`

Like the arithmetic comparison functions, `or` can receive more that two arguments like in this expression: `(or true false true false)`.

`or` can also receive one argument: `(or true)` is perfectly valid.
or can even receive zero argument: `(or)` is valid. A bit silly but valid.

Now let's explore the behaviour of `or`. When all the arguments are boolean, the rule is simple: `or` returns `true` when one of its arguments is `true` and `false` when all of its arguments are `false`.

`(or true)` is `true`.
`(or false)` is `false`.

`(or true false)` is `true` because the first argument is `true`.
`(or false false)` is `false` because all the arguments are `false`.

`(or true false true)` is `true` because one of the arguments is `true`.
`(or false false false)` is `false` because all the arguments are `false`.

`(or (> 5 3) (> 5 19))` is `true` because the first expression is `true`: 5 is greater than 3.

`(or (> 5 13) (> 5 10))` is `false` because all the arguments are `false`: 5 is not greater than 13 and 5 is also not greater than `10`.
`(or (pos? 12) (neg? 4))` is `true` because the first expression is `true`: 12 is positive.

Finally, let's take a look at the behaviour of `or` when it receives no arguments: `(or)` is a weird situation that falls under the case of ***"all the arguments are false"*** in a special way because indeed none of the arguments are true. Therefore, `(or)` is `nil`.

Like in other programming languages, `or` is smart enough not to evaluate arguments when it is not necessary. `or` goes over its arguments from left to right. As soon as it encounters an expression that is `truthy`, it stops the evaluation and returns this truthy value.

For instance, in the expression `(or (pos? 12) (neg? 4))`, `(pos? 12)` is `true`. Therefore `(neg? 4)` will not be evaluated at all.

non-boolean arguments

So far, we have seen how `or` behaves when all its arguments are boolean.

Like with `not` and `and`, the arguments of `or` are not required to be booleans. You can pass any value or expression to `or`.

`or` goes over its arguments from left to right and returns the first truthy argument it encounters:

- When one of the arguments is truthy (neither `false` nor `nil`), its value is immediately returned (without evaluating the remaining arguments).
- When all the arguments are falsy, the value of the last argument is returned.

`(or true false true)` is `true` because the first truthy argument is `true`.

`(or false nil)` is `nil` because all the arguments are falsy and the last arguments is `nil`.

`(or 8 true false)` is 8 because because the first truthy argument is 8.
`(or false nil false)` is `false` because all the arguments are falsy and the last argument is false.

Scenario	**Behavior**	**Example**	**Result**
0 argument	return nil	`(or)`	`nil`
1 argument	return true if the argument is truthy	`(or true)`	`true`
2 arguments	return true if one of the arguments is truthy	`(or true false)`	`true`
any number of arguments	return true if one of the arguments is truthy	`(or true true false true)`	`true`
non-boolean values	return false if all arguments are falsy. return the first truthy argument when one of the arguments is truthy	`(or 4 5)`	4

Table 6.3
The behaviour of the `or` form according to different types and numbers of arguments.
For each case, the behavior of `or` is explicitly stated and illustrated by a concrete example with the result of the evaluation of the expression in the example

Let's practice now on combining `or` with `not`.

`(not (or true false)) is false because (or true false) is true`

`(or true (not false))` is `true` because the first argument is `true`.

`(or (= 2 4) (not (= 4 4)))` is `false` because all the arguments are `false`.

`(or (not (= 2 2)) (not (= 4 4)))` is `false` because all the arguments are `false`.

Summary

In this Lesson, we have met our first non-numbers friends: two of them were booleans (true and false) and the other one was a special creature named `nil`. We have seen that the only falsy values in Clojure are `nil` and `false`. This was important to clarify the difference between *falsy* and `false` and between *truthy* and `true`, because in Clojure, not only booleans but any value can be passed as an argument to a logic form.

We have explored how to write logic operations in Clojure using the `not` form, the `and` form and the `or` form. We have seen that the logic forms follow the same syntax as arithmetic operation and comparison forms. The operation always appear before the operands. As a consequence we can pass more than two arguments to a logic form without having to repeat the symbol form.

At this point, you have a solid basis for the manipulation of numbers and booleans in Clojure.

The next Lesson will conclude Unit 1 by exploring in more details the structure of complex nested expressions in Clojure.

Try this

Ex 1:

Write the following Clojure expressions in Java notation

```
(and true false)
(or true true false)
(not true)
```

Ex 2:

Write the following Clojure expressions in Java notation

```
(and (> 2 3) (= 3 3))
(or (> 12 3) (< 3 3))
(not (> 1 3))
```

Ex 3:

Convert the following Java expressions into Clojure expressions:

```
true && false
true || false || true
!true
```

Ex 4:

Convert the following Java expressions into Clojure expressions:

```
(2 > 3) && (5 == 6)
(2 < 13) || (5 >= 6)
!(2 < 9)
```

Ex 5:

Convert the following Java expressions into Clojure expressions:

```
(2 > 3) && (5 == 6) && (3 <= 9)
(2 < 13) || (5 >= 6) || (8 > 9)
!!(10 < 9)
```

Answers

Ex 1:

Write the following Clojure expressions in Java notation

```
(and true false)
true && false

(or true true false)
true || true || false

(not true)
!true
```

Ex 2:

Write the following Clojure expressions in Java notation

```
(and (> 2 3) (= 3 3))
2 > 3 && 3 == 3
```

```
(or (> 12 3) (< 3 3))
12 > 3 || 3 < 3

(not (> 1 3))
!(1 > 3)
```

Ex 3:

Convert the following Java expressions into Clojure expressions:

```
true && false
(and true false)

true || false || true
(or true false true)

!true
(not true)
```

Ex 4:

Convert the following Java expressions into Clojure expressions:

```
(2 > 3) && (5 == 6)
(and (> 2 3) (= 5 6))

(2 < 13) || (5 >= 6)
(or (< 2 13) (>= 5 6))

!(2 < 9)
(not (< 2 9))
```

Ex 5:

Convert the following Java expressions into Clojure expressions:

```
(2 > 3) && (5 == 6) && (3 <= 9)
(and (> 2 3) (= 5 6) (<= 3 9))

(2 < 13) || (5 >= 6) || (8 > 9)
(or (< 2 13) (>= 5 6) (> 8 9))

!!(10 < 9)
(not (not (< 10 9)))
```

Lesson 7

Complex nested expressions

At this point, your mind starts to get used to the wrapping parenthesis in Clojure: simple compound expressions like `(+ 2 3)`, `(* 3 4 5)` and `(or true false)` don't feel as weird as they felt when you start to read this book.

But what about complex nested expressions like `(or (and (> 3 2) (zero? 2)) (not (= 2 3)))`? It probably still makes you a bit of discomfort. That's perfectly fine. The purpose of this Lesson that concludes Unit 1 is to make you feel more comfortable with complex nested expressions.

After reading this lesson, you will be able to:

- Visualize a complex nested expression in a tree diagram
- Describe the structure of a complex nested expression

Visualization

Now that our logical operations portfolio is complete, we can write some complex expressions that combine logic and arithmetic expressions. Once you understand the mechanics of the evaluation of such a complex boolean expression, you will be able to understand the mechanics of the evaluation of any Clojure expression.

Let’s take as an example a nested expression that combines `and`, `or` and `not`.

```
(or (and (> 3 2) (zero? 2)) (not (= 2 3)))
```

This expression looks a bit scary. Let’s reformat it, by having each form argument in a separate line:

```
(or (and (> 3 2)
         (zero? 2))
    (not (= 2 3)))
```

When you develop in a Clojure environment, the auto formatting (also called smart indentation) is done by the editor. In Unit 2, we will see how the REPL handles auto formatting.

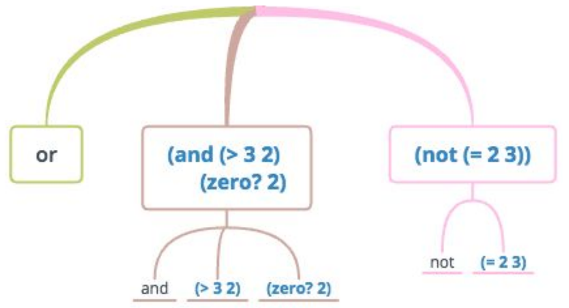
Now that the expression is properly formatted, we start to grasp the structure of the expression: this is an `or` expression with two arguments (which are themselves expressions). The two arguments are:

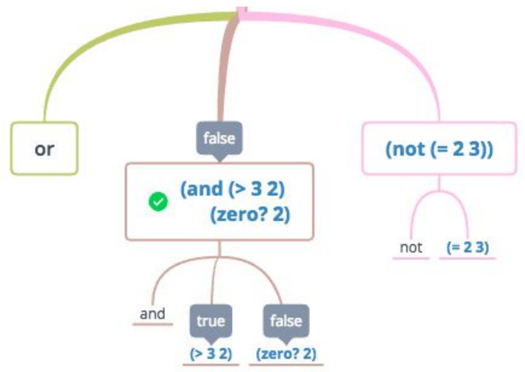
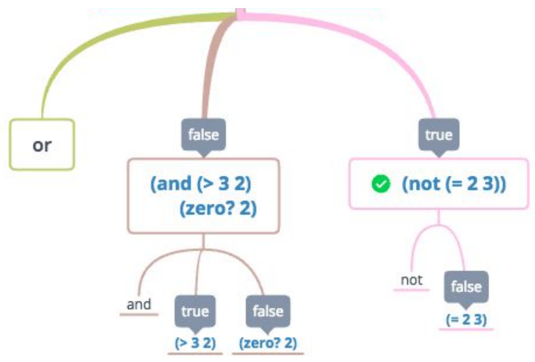
1.
   ```
   (and (> 3 2)
        (zero? 2))
   ```

2.
   ```
   (not (= 2 3))
   ```

In order to understand completely the mechanics of the evaluation of a Clojure expression, we are going to visualize this expression as a tree where at each level of the tree, an expression is a node and its children are the components of the expression: the form symbol and its arguments. This is a recursive process because the arguments of an expression are themselves expressions...

[EDITORIAL: KEYPOINT] When visualizing a nested expression as a tree, each expression becomes a node and its children are the components of the expression: the form symbol and its arguments. This is a recursive process...

Let's start by visualizing this expression as a tree. The root node is:
 	
`(or`
` (and (> 3 2)`
`      (zero? 2))`
` (not (= 2 3)))`
 	
It has three children:
1. The `or` symbol
2. The `and` expression
3. The `not` expression
	 or
	(and (> 3 2) (zero? 2))
	(not (= 2 3))
	and
	(> 3 2)
	(zero? 2)
	not
	(= 2 3)

Let's evaluate the second child which corresponds to the first argument of the `or` expression: this an `and` expression. In order to evaluate it, we need first to evaluate its arguments: `(> 3 2)` is `true` because 3 is greater than 2 `(zero? 2)` is false because 2 is not zero The and expression becomes `(and true false)` which is `false`.	 or false (and (> 3 2) (zero? 2)) (not (= 2 3)) and true (> 3 2) false (zero? 2) not (= 2 3)
Let's evaluate the third child which corresponds to the second argument of the `or` expression: this is a `not` expression. In order to evaluate it, we need first to evaluate its arguments: `(= 2 3)` is false because 2 doesn't equal 3. The not expression becomes `(not false)` which is `true`.	 or false (and (> 3 2) (zero? 2)) true (not (= 2 3)) and true (> 3 2) false (zero? 2) not false (= 2 3)

Finally, we can evaluate the top expression which is the same as `(or false true)` which is `true`.

Now, let's apply the same mechanism to another Clojure expression without drawing all the intermediate steps in such details.

```
(or
 (and (> 3 2)
      (not (pos? -2)))
 (not (= (+ 2 2) 4)))
```

Instead of a tree diagram, let's write an outline in a nested bullet list:

```
1.  or
2.  (and (> 3 2) (not (pos? -2)))
        a.  and
        b.  (> 3 2)
        c.  (not (pos? -2))
              i.    not
             ii.    (pos? -2)
3.  (not (= (+ 2 2) 4))
        a.  not
        b.  (= (+ 2 2) 4)
              i.    =
             ii.    (+ 2 2)
            iii.    4
```

We can also visualize this in a table

	children		analysis	result
`(or (and (> 3 2) (not (pos? -2)) (not (= (+ 2 2) 4)))`	1	`(or`		`(or`
	2	`(and (> 3 2) (not (pos? -2)))`	1. left child `(> 3 2)` is **`true`** 2. right child `(not (pos? -2))` "`not false`" is **`true`**	`true`
	3	`(not (= (+ 2 2) 4))`	1. single child `(= (+ 2 2) 4)` is **`true`**	`false` `)`
			The top `or` expression becomes `(or true false)` which is `true because one of the argument is true`	`true`

Table 7.1 The analysis of a nested clojure expression. We decompose each part of the expression until we are able to evaluate it.

Once you get used to the mechanics of Clojure expressions, you will be able to go through all this steps in your mind very quickly. You will acquire this skill by practicing your Clojure ability through the exercises of this book.

Summary

In this **Lesson,** we learnt how to visualize a complex nested expression as a tree diagram, where each part of the expression becomes a node and its children are the components of the expression. We also exposed in details the mechanics of the evaluation of a complex nested expression.

You have learnt how to manipulate numbers and booleans in Clojure and compose beautiful nested expressions made of many kinds of forms.

At this stage of your Clojure journey, your mind is familiar with the Clojure syntax. Now, you are ready to install Clojure on your machine and write Clojure code for real.

This is a free excerpt of the book. In order to read the full book, please visit the book website.

Try this

Ex 1:

Visualize the following Clojure expressions as a tree:

```
(or
 (and (< 2 2)
      (zero? 2))
 (not (= 6 4)))
```

```
(or
 (and (not (> 3 2))
      (not (neg? 2)))
 (not (= (+ 6 2) 4)))
```

Ex 2:

Outline the following Clojure expressions as a nested bullet list:

```
(or
 (and (< 2 2)
      (zero? 2))
 (not (= 6 4)))
```

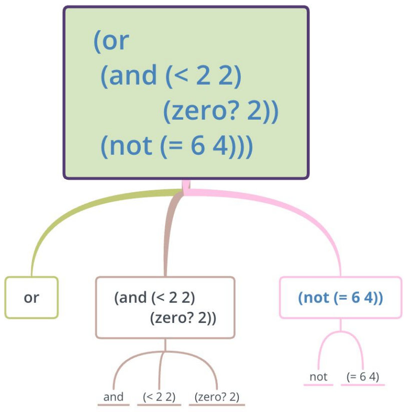
```
(and
 (or (= 2 3)
     (> 4 5))
 (= 4 5)
 (not (> 2 3)))
```

Answers

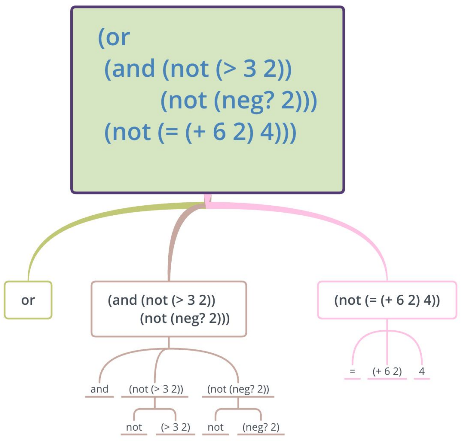
Ex 1:

Visualize the following Clojure expressions as a tree:

```
(or
 (and (< 2 2)
      (zero? 2))
 (not (= 6 4)))
```



```
(or
 (and (not (> 3 2))
      (not (neg? 2)))
 (not (= (+ 6 2) 4)))
```



Ex 2:

Outline the following Clojure expressions as a nested bullet list:

```
(or
 (and (< 2 2)
      (zero? 2))
 (not (= 6 4)))
```

```
    1.  or
    2.  (and (< 2 2) (zero? 2))
            a.  and
            b.  (< 2 2)
            c.  (zero? 2)
    3.  (not (= 6 4))
            a.  not
            b.  (= 6 4)
```

```
(and
 (or (= 2 3)
     (> 4 5))
 (= 4 5)
 (not (> 2 3)))
```

```
    1.  and
    2.  (or (= 2 3) (> 4 5))
            a.  or
            b.  (= 2 3)
            c.  (> 4 5
    3.  (= 4 5)
    4.  (not (> 2 3))
            a.  not
            b.  (> 2 3)
```

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)