ClojureScript Unraveled

Andrey Antukh & Alejandro Gomez

Table of Contents

Chapter 1. Introduction

This chapter will be a first introduction to the clojure ecosystem, and intends to explain the philosophy behind of it.

1.1. The first contact.

ClojureScript is a clojure language that targets javascript and can work in different execution enviroments like browser, nodejs, iojs, nashhorn, and much others.

Unlike other languages that intends to *compile* to javascript (like typescript, funscript or coffeescript) is designed to use the javascript like a bytecode. It embrases the functional programming approach with very safe and consistent defaults.

An other big difference and in my opinion a big advantage over other languages, is that the clojure language is designed to be guest. Is designed as language without own virtual machine that can be easy adaptated to the host differences.

TBD

1.2. The pillars behind the language.

TBD

1.3. Why the javascript host (improve title).

TBD

Chapter 2. The language.

This chapter will be a little introduction to ClojureScript without assumptions about previous knowledge of the Clojure language, providing a quick tour over all the things you will need to know in order to understand the rest of this book.

This is an incomplete list of topics that will be covered in this chapter:

- Syntax
- The base types
- Functions
- Loops & Blocks
- Collection types
- Destructuring
- Namespaces
- Custom Data Types
- Polymorphic constructions
- Interoperability with host
- State management
- Mutability
- Macros
- Truthiness

2.1. First steps with lisp syntax

Invented by John McCarthy in 1958, Lisp is one of the oldest programming languages that is still around. It has evolved into a whole lot of derivatives called dialects and ClojureScript is one of them. It's a programming language written in its own data structures, originally lists enclosed in parenthesis, but Clojure(Script) has evolved the Lisp syntax with more data structures making it more pleasant to write and read.

A list with a function in the first position is used for calling a function in ClojureScript:

```
(+ 1 2 3)
;; => 6
```

In the example above we're applying the addition function **+** to the arguments 1, 2 and 3. ClojureScript allows many unusual characters like **?** or **-** in symbol names which makes it easier to read:

```
(zero? 0)
;; => true
```

For distinguishing function calls and lists, we can quote lists for turning off evaluation. The quoted lists will be treat as data instead of as a function call:

```
'(+ 1 2 3)
;; => (+ 1 2 3)
```

ClojureScript uses more than lists for its syntax, the full details will be covered later but here is an example of the usage of a vector (enclosed in brackets) for defining local bindings:

```
(let [x 1
      y 2
      z 3]
  (+ x y x))
;; => 6
```

This is practically all the syntax we need to know for using not only ClojureScript, but any Lisp. Being written in its own data structures (often referred to as homoiconicity) is a great property since the syntax is uniform and simple; also, code generation via macros is easier than in any language, giving us plenty of power for extending the language to our needs.

2.2. The base data types.

The ClojureScript language has a rich set of data types like most programmig languages. It provides scalar datatypes that will be very familiar for you such as numbers, strings, floats. But, also provides a great amount of others that maybe are not well known such as symbols, keywords, regex, vars, atoms, volatiles…

ClojureScript embrases the host language, and as possible it uses the host provided types. In this case: numbers and strings are used as is and them behaves in same way as in javascript.

2.2.1. Numbers

In *ClojureScript* the numbers includes both: integers and floating points. But, knowing that *ClojureScript* is a guest language that compiles to javascript, having integers is an ilusion. Because the javascript language treats all numbers as floating points values.

Like in any other languages, the numbers in *ClojureScript* are represented in following way:

```
23
+23
-100
1.7
-2
33e8
12e-14
3.2e-4
```

2.2.2. Keywords

Keywords in *ClojureScript* are objecs that always evaluate to themselves. Them are usually used in map data structures for reprensent in a most efficient way to the keys.

```
:foobar
:2
:?
:foo/bar
```

As you can see, the keyword are all prefixed with `:`, but this char is only part of literal syntax and is not part of the name of the object.

You also can create a keyword calling a function **`keyword`**. Do not worry if you do not understand or something is not clear in the following example, the functions are discussed some chapters below.

```
(keyword "foo")
;; => :foo
```

2.2.3. Symbols

The symbols in *ClojureScript* are very very similar to now known **Keywords**. But them instead of evaluating to themselves, are evalutated to something that them refers, that can be function, variables, …

Them are represented with something that not star with a number

```
sample-symbol
othersymbol
f1
```

Do not worry if you do not understand clearly this part, symbols are used un almost all examples and you will have the oportunity to undesarstand them in a practical way, with examples.

2.2.4. Strings

Nothing new we can explain about strings that you do not known. In *ClojureScript* them are work like in any other language. Them are immutable.

And in this concrete case are the same as in javascript:

```
"A example of a string"
```

The pecularity of Strings on *ClojureScript* is due to lisp syntax, and is that you don't need additional syntax for multiline strings:

```
"This is a multiline
      string in ClojureScript."
```

2.2.5. Characters

ClojureScript also has a representation for one character and it has a literal syntax for represent them.

```
\a        ; The lowercase a character
\newline  ; The new line character
```

As its host does not a clear representation for character type, in *ClojureScript* behind the scenes one character is a simple string with one character.

2.2.6. Collections

As usual, the second big step on explaining one language, is explain its collections and collection abstractions. The *ClojureScript* is not an exception in this rule.

ClojureScript comes with great bunch of different collections. The main difference of *ClojureScript* collections with other languages is that them are persistent and immutable.

But before venture of all these (maybe) unknown concepts, we'll go to make a high level overview of existing collection types in *ClojureScript*.

Lists

This is a clasic collection type in lisp languages. *ClojureScript* is not an exception. List is the simplest collection data structure in *ClojureScript*. Lists can contain items of any type, including other collections.

Lists in *ClojureScript* are repsesented with parentheses as its literal syntax:

```
'(1 2 3 4 5)
'(:foo :bar 2)
```

As you can observe, all list examples are prefixed with **`'`** char. This is because lists in lisp like languages are often used for express expressions forms such as function or macro calls. In that case the first item should be a symbol that will evaluate to a something callable and the rest of list elemenents will be a function parameters.

```
(inc 1)
;; => 2

'(inc 1)
;; => (inc 1)
```

As you see, if you will evaluate the **`(inc 1)`** without prefixing it with **`'`** char, it will resolve the **`inc`** symbol to the **inc** function and will execute it with **`1`** as first parameter. Resulting in a **`2`** as return value.

Lists have the pecularity that they are very efficient if you access to it in a sequence mode or access to its first elements but are not very good option if you need random (index) acces to its elements.

Vectors

Like lists, **Vectors** store a series of values, but in this case with very efficient index access to its elements and its elements in difference with list are evaluated in order. Do not worry, in below chapters we'll go depth in details but at this moment is more that enough.

Vectors uses square brakets for the literal syntax, let see some examples:

```
[:foo :bar]
[3 4 5 nil]
```

Like lists, vectors can contain objects of any type, as you can observe the previos example.

Maps

Maps is a collection abstraction that allows store unique keys associated with one value. In other languages are commonly known as hash-maps or dicts. Maps in *ClojureScript* uses a curly braces as literal syntax.

```
{:foo "bar", :baz 2}
{:foobar [:a :b :c]}
```

> Commas are frequently used for separate a key value pair but are completelly optional. In *ClojureScript* syntax, comas are treated like spaces.

Like Vectors, every item in a map literal is evaluated before the result is stored in a map, but the order of evaluation is not guaranteed.

Sets

And finally, **Sets**.

Sets stores in an unordered way zero or more unique items of any type. They, like maps, uses curly braces for its literal syntax with difference that uses a # as leading character:

```
#{1 2 3 :foo :bar}
```

In below chapters we'll go depth in sets and other collection types explained in this chapter.

2.3. Vars

ClojureScript is a mostly functional language and focused in immutability. Becuase of that, it does not has the concept of variables. The most closest analogy to variables are **vars**. The vars are represented by symbols and stores a single value together with metadata.

You can define a var using a **`def`** special form:

```
(def x 22)
(def y [1 2 3])
```

The vars are always top level in the namespace. If you use **`def`** in a function call, the var will be defined at the namespace level.

2.4. Functions

2.4.1. The first contact

It's time of make things happen. In *ClojureScript*, a function are first-class type. It behaves like any other type, you can pass it as parameter, you can return it as value, always respecting the lexical scope. *ClojureScript* also has some features from dynamic scope but this will be discused in other section.

If you want know more about scopes, this wikipedia article[1] is ver extensive and explain very well different types of scope.

As *ClojureScript* is a lisp dialect, it uses the prefix notation for calling a function:

```
(inc 1)
;; => 2
```

The **`inc`** is a function and is part of *ClojureScript* runtime, and **`1`** is a first positional argument for the **`inc`** function.

```
(+ 1 2 3)
```

[1] http://en.wikipedia.org/wiki/Scope_%28computer_science

```
;; => 6
```

The **+** symbol represents a **add** function, in ALGOL type of languages is an operator and only allows two parameters.

The prefix notation has huge advantages, some of them not alwats obvious. *ClojureScript* does not has distinction between a function and operator, everything is a function. The inmediate advantage is that the prefix notation allows an arbitrary number of arguments per "operator". Also, it eliminates per complete the problem of operator precedence.

2.4.2. Defining own functions

The function can be defined with **fn** special form. This is aspect of function definition:

```
(fn [param1 param2]
  (+ (inc param1) (inc param2)))
```

You can define a function and call it in same time (in a single expression):

```
((fn [x] (inc x)) 1)
;; => 2
```

Let start creating named functions. But that is means named function really? Is very simple, as in *ClojureScript* functions are fist-class and behaves like any other value, naming a function is just store it in a var:

```
(def myinc (fn [x] (+ x 1)))

(myinc 1)
;; => 2
```

ClojureScript also offers the **defn** macro as a little sugar syntax for make function definition more idiomatic:

```
(defn myinc
  "Self defined version of `inc`."
  [x]
  (+ x 1))
```

2.4.3. Function with multiple arities

ClojureScript also comes with ability to define functions with arbitrary number of arities. The syntax is almost the same as define standard function with the difference that it has more that one body.

Let see an example, surelly it will explain it much better:

```
(defn myinc
  "Self dined version of parametrized `inc`."
  ([x] (myinc x 1))
  ([x increment]
   (+ x increment)))
```

And there some examples using the previously defined multi arity function. I can observe that if you call a function with wrong number of parameters the compiler will emit an error about that:

```
(myinc 1)
;; => 1

(myinc 1 3)
;; => 4

(myinc 1 3 3)
;; Compiler error
```

> Explaining the "arity" is out of scope of this book, however you can read about that in this wikipedia article[2].

2.4.4. Variadic functions

An other way to accept multiple parameters is defining variadic functions. Variadic functions are functions that will be able accept arbitrary number of arguments:

```
(defn my-variadic-set
  [& params]
  (set params))
```

[2] http://en.wikipedia.org/wiki/Arity

```
(my-variadic-set 1 2 3 1)
;; => #{1 2 3}
```

The way to denone a variadic function is using the **&** simbol prefix on its arguments vector.

2.4.5. Short syntax for anonymous functions

ClojureScript provides a shorter syntax for define anonymos (and almost always one liner) functions using the **#()** reader macro. Reader macros are "special" expressions that in compile time will be transformed to the apropiate language form. In this case to some expression that uses **fn** special form.

```
(def my-set #(set %1 %2))

(my-set 1 2)
;; => #{1 2}
```

The **%1**, **%2**, **%N** are simple markers for parameter positions that are implicitly declared when the reader macro will be interpreted and converted to **fn** expression.

Also, if a function only accepts one argument, you can ommit the number after **%** symbol, the function **#(set %1)** can be written **#(set %)**.

Additionaly, this syntax also supports the variadic form with **%&** symbol:

```
(def my-variadic-set #(set %&))

(my-variadic-set 1 2 2)
;; => #{1 2}
```

2.5. Flow control

ClojureScript has a great different approaches for flow control.

2.5.1. Branching with **if**

Let start with a basic one: **if**. In *ClojureScript* the **if** is an expression and not an statement, and it has three parametes: first one the condition expression, the second one a expression that will evalute if a condition expression will evalute in a logical true, and the third one will evaluated otherwise.

```
(defn mypos?
  [x]
  (if (pos? x)
    "positive"
    "negative"))

(mypos? 1)
;; => "positive"

(mypos? -1)
;; => "negative"
```

If you want do more that one thing in one of two expressions, you can use block expression **do**, that is will explained in next section.

2.5.2. Branching with **cond**

Sometimes, the **`if`** expression can be slightly limited because it does not have the "else if" part for add more that one condition. The cond comes to the rescue.

With **cond** expression, you can define multiple conditions:

```
(defn mypos?
  [x]
  (cond
    (> x 0) "positive"
    (< x 0) "negative"
    :else "zero"))

(mypos? 0)
;; => "zero"

(mypos? -2)
;; => "negative"
```

Also, cond has an other form, called **condp**, that works very similar to the simple **cond** but looks more cleaner when a predicate is always the same for all conditions:

```
(defn translate-lang-code
  [code]
  (condp = (keyword code)
    :es "Spanish"
    :en "English"
```

```
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

2.5.3. Branching with **case**

The **case** branching expression has very similar use case that our previous example with **condp**. The main difference is that, case always uses the **=** predicate/function and its branching values are evaluated at compile time. This results in a more prerformant form that **cond** or **condp** but has the disadvantage of that the condition value should be a static value.

Let see the same example as previous one but using **case**:

```
(defn translate-lang-code
  [code]
  (case code
    "es" "Spanish"
    "es" "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

2.6. Locals, Blocks and Loops

2.6.1. Locals

ClojureScript does not has the variables concepts, but it does have locals. Locals as per usual, are immutable and if you try mutate them, the compiller will throw an error.

The locals are defined with **let** expression. It starts with a vector as first parameter followed by arbitrary number of expresions. The first parameter should contain a arbitrary number of pairs that starts with a binding form followed of an expression whose value will be bound to this new local for the remainer of the let expression.

```
(let [x (inc 1)
      y (+ x 1)]
  (println "Simple message from body of let")
  (* x y))
;; => 6
```

2.6.2. Blocks

The blocks in *ClojureScript* can be done using the **`do`** expression and is ususally used for side effects, like printing something in console or write a log in a logger. Something for that the return value is not necesary.

The **`do`** expression accept as parameter an arbitrary number of other expressions but return the return value only from the last one:

```
(do
   (println "hello world")
   (println "hola mundo")
   (+ 1 2))
;; => 3
```

The **`let`** expression, explained just in previous section, the body is very similar to the **`do`** expression. In fact, it is called that is has an implicit **`do`**.

2.6.3. Loops

The functional approach of *ClojureScript*, this causes that it does not have standard, well known statements based loops. The loops in clojurescript are handled using recursion. The recursion sometimes requires additional thinking about how model your problem in a slightly different way than imperative languages.

Also, many of the common patterns for which **`for`** is used in other languages are achieved through higher-order functions.

Looping with loop/recur

Let’s take a look at how to express loops using recursions with the **`loop`** and **`recur`** forms. **`loop`** defines a possibly empty list of bindings (notice the symmetry with **`let`**) and **`recur`** jumps execution after the looping point with new values for those bindings.

Let’s see an example:

```
(loop [x 0]
   (println "Looping with " x)
   (if (= x 2)
     (println "Done looping!")
     (recur (inc x))))
;; => "Looping with 0"
;; => "Looping with 1"
;; => "Looping with 2"
;; => "Done looping!"
```

In the above snippet, we bind the name **`x`** to the value **`0`** and execute the body. Since the condition is not met the first time is run we **`recur`**, incrementing the binding value with the **`inc`** function. We do this once more until the condition is met and, since there aren't more **`recur`** calls, exit the loop.

Note that **`loop`** isn't the only point we can **`recur`** too, using **`recur`** inside a function executes the body of the function recursively with the new bindings:

```
(defn recursive-function [x]
   (println "Looping with" x)
   (if (= x 2)
     (println "Done looping!")
     (recur (inc x))))

(recursive-function 0)
;; => "Looping with 0"
;; => "Looping with 1"
;; => "Looping with 2"
;; => "Done looping!"
```

Replacing for loops with higher-order functions

In imperative programming languages is common to use **`for`** loops for iterating over data and transforming it, usually the intent being one of the following:

- Transform every value in the iterable yielding another iterable
- Filter the elements of the iterable by a certain criteria
- Convert the iterable to a value where each iteration depends on the result from the previous one
- Run a computation for every value in the iterable

The above actions are encoded in higher-order functions and syntactic constructs in ClojureScript, let's see an example of the first three.

For transforming every value in a iterable data structure we use the **`map`** function, which takes a function and a sequence and applies the function to every element:

```
(map inc [0 1 2])
;; => (1 2 3)
```

For filtering the values of a data structure we use the **`filter`** function, which takes a predicate and a sequence and gives a new sequence with only the elements that returned **`true`** for the given predicate:

```
(filter odd? [1 2 3 4])
;; => (1 3)
```

Converting an iterable to a value accumulating the intermediate result in every step of the iteration can be achieved with **`reduce`**, which takes a function for accumulating values, an optional initial value and a collection:

```
(reduce + 0 [1 2 3 4])
;; => 10
```

`for` sequence comprehensions

In ClojureScript the **`for`** construct isn't used for iteration but for generating sequences, an operation also known as "sequence comprehension". It offers a small domain specific language for declaratively building lazy sequences.

It takes a vector of bindings and a expression and generates a sequence of the result of evaluating the expression, let's take a look at an example:

```
(for [x [1 2 3]]
  [x x])
;; => ([1 1] [2 2] [3 3])
```

It supports multiple bindings, which will cause the collections to be iterated in a nested fashion, much like nesting **`for`** loops in imperative languages:

```
(for [x [1 2 3]
```

```
      y [4 5]]
  [x y])
;; => ([1 4] [1 5] [2 4] [2 5] [3 4] [3 5])
```

We can also follow the bindings with three modifiers: `:let` for creating local bindings, `:while` for breaking out of the sequence generation and `:when` for filtering out values.

Here's an example of local bindings using the `:let` modifier, note that the bindings defined with it will be available in the expression:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]]
  z)
;; => (5 6 6 7 7 8)
```

We can use the `:while` modifier for expressing a condition that, when it is no longer met, will stop the sequence generation. Here's an example:

```
(for [x [1 2 3]
      y [4 5]
      :while (= y 4)]
  [x y])
;; => ([1 4] [2 4] [3 4])
```

For filtering out generated values we use the `:when` modifier like in the following example:

```
(for [x [1 2 3]
      y [4 5]
      :when (= (+ x y) 6)]
  [x y])
;; => ([1 5] [2 4])
```

We can combine the modifiers shown above for expressing complex sequence generations or more clearly expressing the intent of our comprehension:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]
      :when (= z 6)]
```

```
  [x y])
;; => ([1 5] [2 4])
```

When we outlined the most common usages of the **for** construct in imperative programming languages we mentioned that sometimes we want to run a computation for every value in a sequence, not caring about the result. Presumably we do this for achieving some sort of side-effect with the values of the sequence.

ClojureScript provides the **doseq** construct, which is analogous to **for** but executes the expression discarding the resulting values and returns **nil**.

```
(doseq [x [1 2 3]
        y [4 5]
       :let [z (+ x y)]]
  (println x "+" y "=" z))
;; => "1 + 4 = 5"
;; => "1 + 5 = 6"
;; => "2 + 4 = 6"
;; => "2 + 5 = 7"
;; => "3 + 4 = 7"
;; => "3 + 5 = 8"
;; => nil
```

2.7. Collection types

2.7.1. Immutable and persistent

We mentioned before that ClojureScript collections are persistent and immutable but didn’t explain what we meant.

An immutable data structure, as its name suggest, is a data structure that can not be changed. In-place updates are not allowed in immutable data structures.

A persistent data structure is a data structure that returns a new version of itself when transforming it, leaving the original unmodified. ClojureScript makes this memory and time efficient using an implementation technique called structural sharing, where most of the data shared between two versions of a value is shared and transformations of a value are implemented by copying the minimal amount of data required.

Let’s see an example of appending values to a vector using the **conj** (for "conjoin") operation:

```
(let [xs [1 2 3]
      ys (conj xs 4)]
  (println "xs:" xs)
  (println "ys:" ys))
;; => xs: [1 2 3]
;; => ys: [1 2 3 4]
;; => nil
```

As you can see, we derived a new version of the **xs** vector appending an element to it and got a new vector **ys** with the element added.

For illustrating the structural sharing of ClojureScript data structures, let's compare whether some parts of the old and new versions of a data structure are actually the same object with the **identical?** predicate. We'll use the list data type for this purpose:

```
(let [xs (list 1 2 3)
      ys (cons 0 xs)]
  (println "xs:" xs)
  (println "ys:" ys)
  (println "(rest ys):" (rest ys))
  (identical? xs (rest ys)))
;; => xs: (1 2 3)
;; => ys: (0 1 2 3)
;; => (rest ys): (1 2 3)
;; => true
```

As you can see in the example, we used **cons** (construct) to prepend a value to the **xs** list and we got a new list **ys** with the element added. The **rest** of the **ys** list (all the values but the first) are the same object in memory that the **xs** list, thus **xs** and **ys** share structure.

2.7.2. The sequence abstraction

2.7.3. Collections in depth

Lists

Vectors

Maps

Sets

2.8. Destructuring

TBD

2.9. Namespaces

2.9.1. Defining a namespace

Namespaces is a clojurescript's fundamental unit of code modularity. Are analogous to Java packages or Ruby and Python modules, and can be defined with **ns** macro. Maybe if you are touched a little bit of clojurescript source you have seen something like this at begining of the file:

```
(ns myapp.core
  "Some docstring for the namespace.")

(def x "hello")
```

Namespaces are dynamic and you can create one in any time, but the convention is having one namespace per file. So, the namespace definition usually is at begining of the file followed with optional docstring.

Previously we have explained the vars and symbols. Every var that you are defines will be associated with one namespace. If you do not define a concrete namespace, the default one called "user" will be used:

```
(def x "hello")
;; => #'user/x
```

2.9.2. Loading other namespaces

It's ok, definining a namespace and vars in it is really easy, but it is not very usefull if we can't use them from other namespaces. For this purpose, the **ns** macro also offers a simple way to load other namespaces.

Observe the following:

```
(ns myapp.main
  (:require myapp.core
            clojure.string))

(clojure.string/upper-case myapp.core/x)
;; => "HELLO"
```

As you can observe, we are using fully qualified names (namespace + var name) for access to vars and functions from different namespaces.

It is ok, we not can access to other namespaces but is very boring always write the complete namespace name for access to its vars and functions. It will be specially uncomfortable if a namespace name is very large. For solve that, you can use the **`:as`** directive for create an additional (usually more shorter) alias to the namespace. Let see the how it can be done:

```
(ns myapp.main
  (:require [myapp.core :as core]
            [clojure.string :as str]))

(str/upper-case core/x)
;; => "HELLO"
```

Additionaly, *ClojureScript* offers a simple way to refer specific vars or functions from concrete namespace using the **`:refer`** directive.

The **`:refer`** directive has two possible arguments: **`:all`** keyword or a vector of symbols that will refer to vars in the namespace. With **`:all`** we are indicating that we want refer all public vars from the namespace and with vector we can specify the concrete subset of vars that we want.

```
(ns myapp.main
```

```
  (:require [myapp.core :refer :all]
            [clojure.string :refer [upper-case]]))
```

And finally, we should know that everything that located in the **`cljs.core`** namespace is automatically loaded and you should not require it explicitly. But sometimes you want declare vars that will clash with some other defined in **`cljs.core`** namespace. For it, the **`ns`** macro offers an other directive that allows exclude concrete symbols and prevet them to be automaticaly loaded.

Observe the following:

```
(ns myapp.main
  (:refer-clojure :exclude [min]))

(defn min
  [x y]
  (if (> x y)
    y
    x))
```

The **`ns`** macro also has other directives for loading host clases (**`:import`**) and macros (**`:refer-macros`**), but them are explained in posterior sections.

2.10. Abstractions and Polymorphism

I’m sure that in more that in one time you have found in this situation: you have defined a great abstraction (using interfaces or something similar) for your "bussines logic" and you have found the need to deal with an other module over which you have absolutelly no control, and you probably was thinking in create adapters, proxies and other approaches that will implies a great amount of additional complexity.

Some dynamic languages allows "monkey-patching", languages where the classes are open and any method can be defined and redefined at any time. Also, is very known that this technique is a very bad practice.

We can not trust languages that allows that when importing third party libraries, can silently overwrite methods that you are using and expecting a concrete behavior.

This symptoms denotes a commonly named: "Expression problem".

TODO: add link to expression problem description

2.10.1. Protocols

The *ClojureScript* primitive for define "interfaces" are called Protocols. A protocol consists in a name and set of functions. All functions have at least one argument corresponding to the **`this`** in javascript or **`self`** in Python.

Protocols provides a type based polymorphism, and the dispatch is always done by the first argument previously mentioned as **`this`**.

A protocol looks like this:

```
(ns myapp.foobar)

(defprotocol IUser
  "A common abstraction for an User type."
  (full-name [x] "Get a full name of the user."))
```

> the "I" prefix is very common for make clear separation of protocols and types. In clojute comunity it there many dispare optionions about the use of the "I" prefix. In our opinion is an acceptable solution for avoid name clashing and confusions.

From the user perspective, protocol functions are simple and plain functions defined in the namespace where the protocol is defined. As you can intuit, this makes protocols completelly namespaces and avoid any accidental clashing between implemented protocols for same type.

> Inside the clojure community the convention for protocol functions is treat them as private api and not expose them to the general public api.

Extending to existing types

TBD

Protocols introspection

TBD

Edge cases

TBD

Participate in ClojureScript abstractions

TBD

2.10.2. Multimethods

TBD

2.11. Data types

Until, now, we have used maps, sets, lists and vectors for represent our data. And in most cases is a really great aproach for do it. But some times we need define our own types and in this book we will call them **datatypes**.

A datatype provides the following:

- A unique host backed type, either named or anonymous.
- Explicitly declared structure using fields or closures.
- Implement concrete abstractions.
- Map like behavior (via records, see below).

2.11.1. Deftype

The most low level construction in *ClojureScript* for create own types, is the **`deftype`** macro. For demostration we will define a type called **`User`**:

```
(deftype User [firstname lastname])
```

Once the type has beed defined, we can create an instance of our **`User`**:

```
(def user (User. "Triss" "Merigold"))
```

And its fields can be accesset using the prefix-dot notation:

```
(.-firstname user)
;; => "Triss"
```

Types defined with deftype (and posteriory with defrecord) creates a host backed class like object associated to the current namespace. But it has some peculiarities when we intend to use or import it from other namespace. The types in *ClojureScript* should be imported with **`:import`** directive of **`ns`** macro:

```
(ns myns.core
  (:import otherns.User))

(User. "Cirilla" "Fiona")
```

For convenience, *ClojureScript* also defines a constructor function caled **`→User`** that can be imported with the common way using **`:require`** directive.

We personally do not like this type of functions, and we prefer define own constructors, with more idiomatic names:

```
(defn user
  [firstname lastname]
  (User. firstname lastname))
```

And use it in our code instead of **`→User`**.

2.11.2. Defrecord

The record is a slightly higher level abstraction for define types in *ClojureScript* and should be prefered way to do it.

As we know, *ClojureScript* tends to use plain data types how are the maps but in most cases we need have a named type for represent the entities of our application. Here come the records.

A record is a datatype that implements a map protocols and therefore can be used like any other map. And since records are also proper types, they support type-based polymorphism through protocols.

In summary: with records, we have the best of both worlds, maps that can play in in different abstractions.

Let start defining the **`User`** type but using records:

```
(defrecord User [firstname lastname])
```

It looks really similar to deftype syntax, in fact, it uses deftype behind the scenes as low level primitive for defining types.

Now, look the difference with raw types for access to its fields:

```
(def user (User. "Yennefer" "of Vengerberg"))

(:username user)
;; => "Yennefer"

(get user :username)
;; => "Yennefer"
```

As we mention previously, records are maps and acts like tham:

```
(map? user)
;; => true
```

And like maps, tham support extra fields that are not initially defined:

```
(def user2 (assoc user :age 92))

(:age user2)
;; => 92
```

As we can see, the **`assoc`** function works as is expected and return a new instance of the same type but with new key value pair. But take care with **`dissoc`**, its behavior with records is slightly different that with maps; it will return a new record if the field being dissociated is an optional field, but it will return a plain map if you dissociate the mandatory field.

An other difference with maps is that records does not acts like functions:

```
(def plain-user {:username "Yennefer", :lastname "of Vengerberg"})

(plain-user :username)
;; => "Yennefer"

(user :username)
;; => user.User does not implements IFn protocol.
```

The **`defrecord`** macro like the **`deftype`**, for convenience esposes **`→User`** function, but with additional one **`map→User`** constructor function. We have the same opionon about that constructors that with deftype defined ones: we recommend define own instead of use that ones. But as they exists, let see how they can be used:

```
(def cirilla (->User "Cirilla" "Fiona"))
(def yen (map->User {:firstname "Yennefer"
                     :lastname "of Vengerberg"}))
```

2.11.3. Implement protocols

Both type definition primitives that we have seen until now allows inline implementations for protocols (explained in previous section). Let start define one for example purposes:

```
(defprotocol IUser
  "A common abstraction for work with user types."
  (full-name [_] "Get the full name of the user."))
```

Now, you can define a type with inline implementation for an abstraction, in our case the **`IUser`**:

```
(defrecord User [firstname lastname]
  IUser
  (full-name [_]
    (str firstname " " lastname)))

;; Create an instance.
(def user (User. "Yennefer" "of Vengerberg"))

(full-name user)
;; => "Yennefer of Vengerberg"
```

2.11.4. Reify

The **`reify`** macro lets you create an anonymous types that implement protocols. In difference with deftype and defrecord, it does not has accessible fields.

This is a way how we can emulate an instance of user type and that plays well in **`IUser`** abstraction:

```
(defn user
  [firstname lastname]
  (reify
```

```
    IUser
    (full-name [_]
      (str firstname " " lastname))))

(def yen (user "Yennefer" "of Vengerberg"))
(full-name user)
;; => "Yennefer of Vengerberg"
```

The real purpose of reify is create anonymous types that plains in a concrete abstractions but you do not want a type in self.

2.12. Host interoperability

TBD

2.13. State management

TBD

2.14. Mutability

TBD

2.15. A little overview of macros

TBD

2.16. Truthiness

TBD

Chapter 3. Tooling & Compiler

This chapter will cover a little introduction to existing tooling for making things easy when developing using ClojureScript. It will cover:

- Using the repl
- Leiningen and cljsbuild
- Google Closure Library
- Modules
- Unit testing
- Library development
- Browser based development
- Server based development

Unlike the previos chapter, this chapter intends to tell different histories not all related to each other.

3.1. Working with the REPL

TBD

3.2. Getting started with Compiler

TBD

3.3. Build & Dependency management tools

3.3.1. Getting started with leiningen.

TBD

3.3.2. Getting started with boot.

TBD

3.4. The Closure Library

TBD

3.5. Browser based development

TBD

3.5.1. Using third party javascript libraryes

TBD

3.5.2. Modularizing your code

TBD

3.6. Developing a library

TBD

3.7. Unit testing

TBD

Chapter 4. Mixed Bag

This chapter will cover miscelaneous topics that are not classified in the previous ones. This is a "catchall" section and will touch a bunch of heterogeneus topics like:

- Async primitives using *core.async* library.
- Working with promises.
- Error handling using *cats* library.
- Pattern matching with *core.match* library.
- Web development using Om library.
- Share code betwen clojure and clojurescript.

4.1. Async primitives using core.async.

TBD

4.2. Working with promises.

TBD

4.3. Error handling using monads and Cats.

TBD

4.4. Pattern matching using core.match.

TBD

4.5. Web development with Om and React.

TBD

4.6. Writing libraries that shares code betwen Clojure and ClojureScript.

TBD