Introduction to Clojure

Mark Armstrong

November 25, 2020

Contents

1 Introduction

These notes were created for, and in some parts **during**, the lecture on November 6th and the following tutorials.

2 Motivation

:TODO:

3 Getting Clojure

For a quick start with Clojure, you can use repl.it.

For instructions on installing Clojure, see the Clojure getting started guide.

For the purposes of testing in Docker, we will use the build tool Leiningen.

That said, we do not assume that a "project" is created for the homeworks and assignments. As with Scala, we will assume that your Clojure code is a *script*, to be run in the REPL.

Specifically, we will run your code by starting a REPL with `lein repl`, and then "dumping" your code into that REPL.

If you want to check how your code runs with `lein repl` on your own system, you can run `cat my-code-file.clj | lein repl`. This is the command the Docker image will use.

4 The syntax and (most of) the semantics of Clojure

The syntax of Lisps such as Clojure tend to be extremely minimal. For today at least, we will work with a subset of the language described by the following grammar, which is sufficient for a fair amount of programming.

```
⟨expr⟩ ::= number
         | "nil"
         | ⟨list⟩
         | ⟨array⟩
         | symbol

⟨list⟩ ::= "(" {⟨expr⟩} ")"

⟨array⟩ ::= "[" {⟨expr⟩} "]"
```

For example, the following are all Clojure expressions.

- Integers.

```
2
-1
```

- Symbols

```
:symbols
:a
:b
```

- Lists (note the `'` or `quote` usage; we will explain it below in 6.)
  - Which are heterogeneous.

```
;; Lists
()
'(1 2 3)
'(:a 2 3)
(quote (1 2 3))
'((1 2) (3 4))

(first '(1 2 3)) ;; => 1
                 ;; first is often called car in other
                 ↪ Lisps.
                 ;; Standing for "Contents of Address
                 ↪ part of Register",
                 ;; referring to the memory layout used
                 ↪ in the original implementation of
                 ↪ Lisp.
(rest '(1 2 3)) ;; => (2 3)
                ;; rest is often called cdr in other
                ↪ Lisps.
                ;; (Pronounced "could-er".)
                ;; Standing for "Contents of Decrement
                ↪ part of Register".
```

- Arrays
  - Also heterogeneous.

```
;; Arrays
[1 2 3]
[:a 1 2]
[]
```

- Sets and maps.
  - Which are also heterogeneous.

  ```
  #{1 2 3}

  {:key 1, "my key" :a_value}
  ```

Clojure programs are written as lists, with the head of the list being the *operator* and the tail of the list being the *operands*. The (regular) semantics of Clojure expressions can be described in just two lines; to evaluate a list,

1. evaluate each element of the list, and then
2. apply the operands to the operator.

By default, Clojure does use call-by-value semantics.

For instance,

```
; (1 + 2) * max(3,4)
(* (+ 1 2)     ; (+ 1 2) evaluates to 3
   (max 4 3)   ; (max 4 3) evaluates to 4
   )           ; (* 3 4) evaluates to 12

(+ 1 2 3 4
   5 6 7 8)
(+ 1)
```

5 Special forms; the `defn`, `def` and `fn` forms

When or if the evaluation rules of Clojure given above prove too limiting, Clojure allows for "special forms" (pieces of syntax handled differently by the compiler) to implement constructs.

5.1 `defn`

The first of these we will consider is the `defn` form, which can be read "define function".

For instance, here is code which defines two methods, called `square` and `sum_of_squares`, and then calls `sum_of_squares` with arguments `2` and `3`.

```
(defn square [x] (* x x))

(defn sum-of-squares [x y]
  (+ (square x)
     (square y)))

(sum-of-squares 2 3)
```

Functions can be defined as having different arities. All the definitions do have to be packaged together as below.

```
; Define them by "brute force"
(defn sum-of-squares
  ([x y]   (+ (square x) (square y)))
  ([x y z] (+ (square x) (square y) (square z))))

(sum-of-squares 2 3)
(sum-of-squares 2 3 4)
```

They can even accept any number of arguments by the use of an ampersand `&`; an `&` followed by a name indicates the function takes any number of additional arguments, which are gathered into a list given that name.

```
;; sum-of-squares-variadic takes any number of arguments,
;; collected into the list xs.
;; (If we wanted to force there to be some arguments,
;; we could put those arguments before the &.)
(defn sum-of-squares-variadic [& xs]
 (if (empty? xs)
   ;; If xs is empty, the sum is 0.
   0
   ;; Otherwise, square the first element of xs, and add it
   ;; to the result of applying sum-of-squares-variadic
   ;; to the rest of the list.
   (+ (square (first xs))
      ;; Note that we need to use apply here as (rest xs) is a
       ↪  list,
      ;; not separate arguments.
      ;; (apply f (x1 x2 … xn)) is equivalent to (f x1 x2 …
       ↪  xn)
      ;; (It also works for arrays of arguments, not just
       ↪  lists.)
```

```
    (apply sum-of-squares-variadic (rest xs)))))

(sum-of-squares-variadic 1 2)
(sum-of-squares-variadic 1 2 3 4 5)
(sum-of-squares-variadic)
```

5.2 `defn` is `def` plus `fn`

The `defn` form can be thought of as the combination of the `def` and `fn` forms. The `def` form defines named values.

```
(def my-favourite-number 16)
```

The `fn` form defines *anonymous* functions. We apply this one right away to see its result.

```
((fn [x] (+ x 1)) my-favourite-number)
```

There is a shorthand for the `fn` syntax; `(fn [args] body)` can be replaced by `#(body)`, with a small change to `body`. There is no specification of argument names with the `#()` form, so they need to be replaced in `body` with *positional argument names*. A `%` is used if the function has a single parameter, `%1`, `%2`, etc. if the function has multiple parameters, and `%&` if it is variadic. So the above can be written

```
(#(+ % 1) my-favourite-number)
```

(Note; nesting of the `#()` form is not allowed, because the positional parameter names would become ambiguous.)

Of course, given `def` and `fn`, we don't need `defn`; we could just write

```
(def add-one (fn [x] (+ x 1)))
```

We've then given the anonymous function a name we can use later.

```
(add-one my-favourite-number)
```

But `defn` is more convenient.

5.3 Scope and the `let` form

The `def` form creates a new *global* binding. This means even if not used at the "top-level", the binding can be seen anywhere.

```clojure
(defn get-the-number []
  (def the-number 3)
  the-number)

(get-the-number)

the-number ;; Still in scope
```

The same applies for `defn`.

```clojure
(defn do-the-thing []
  (defn the-thing [] (+ 1 1))
  (the-thing))

; (the-thing) ;; After definition, the-thing is not yet
↪  defined.

(do-the-thing)

(the-thing) ;; But after execution, it's been bound.
```

In contrast, the `let` form creates a new *local* binding.

```clojure
(defn get-secret-symbol []
  (let [get-symbol (fn [] :secret)]
    (get-symbol)))
```

6 The quote, `'`

Another special form is `quote`, which is used when you want to interpret a list as *data* instead of as a function invokation.

```clojure
; call function +
(+ 1 2 3)

; create a list
(quote (+ 1 2 3))

; syntactic sugar; just prepend a ' to the front of the list
'(+ 1 2 3)
```

7 Conditional forms

The form `if` acts as you would expect; it takes 3 arguments, the first of which is checked for truth, and if it is true, the second argument is evaluated. Otherwise the third argument (if present) is evaluated.

```
(if (> 7 6) "Yes!" "No!")
(if (< 7 6) "Yes!" "No!")

; The else is optional.
(if (= 7 6) "Not okay!")
(if (not= 7 6) "Okay!")
```

All values in Clojure are “true” except for `nil` and `false`.

```
(if 0 "1")
(if () "2")
(if nil "3")
(if false "4")
(if true "5")
```

If you only want a “then” branch for your “if”, then it’s best to use the `when` form.

```
(when true "hello")
```

:TODO: explain

```
(when true (println "Hello!") (println "Goodbye!"))
```

For a more switch-statement like form, use `cond`. Since any value (that is not `false` or `nil`) is truthy, we can include an “else” branch by putting a value as the condition. By convention, the symbol `:else` is used.

```
(cond (> 2 5) "2 is greater"
      (< 2 5) "5 is greater"
      :else "They're the same.")
```

8 `do`, for sequential computation

If we put multiple expressions in sequence, the “result” will usually be the value of the last expression. For instance, in a function definition:

```clojure
(defn add [x]
  (println "hello")
  (+ x x))

(add 5)
```

In some cases, we cannot put a sequence of expressions where we might want one. For instance, when using an `if`, the `then` branch has to be a single expression. To get around this, there is a `do` form for sequencing.

```clojure
(if true
  (do (println "First")
      (println "Second")))
```

There are more complex forms for repeating the same expressions several times;

- `dotimes` for evaluating a certain number of times,

  ```clojure
  (dotimes [i 3] ;; The name is an iterator
    (println i))
  ```

  and

- `doseq` for evaluating once for each element in a sequence.

  ```clojure
  (doseq [i '(1 2 3)] (println i))
  ```

  or iterating over multiple sequences

  ```clojure
  (doseq [i '(1 2 3) j '(:a :b :c)] (println i j))
  ```

9 Side notes

9.1 Partial application

Partial application would be implemented as a function returning a function, and invokation of such a function would look like

```clojure
((f 3) 2)
```