Introduction to Clojure

Mark Armstrong

November 9, 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.

Specific instructions on versions may come later, once a Docker image is decided upon for Clojure.

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.

```
2
-1

:symbols

()
'(1 2 3)
(quote (1 2 3))
'((1 2) (3 4))

[1 2 3]
[]
```

```
;; There are sets as well (unordered collections)
#{1 2 3}

;; Maps as well
{: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.

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.

```
; 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)

; Or define a "variadic" function which takes a list of
↪  arguments
(defn sum-of-squares [x & rest]
  (* (square x) ;; TODO
     ))

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

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.

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

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.

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

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

; syntactic sugar
'(+ 1 2 3)
```

7 Conditional forms

8 `do`, for sequential computation

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

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