Introduction to Clojure

Mark Armstrong

November 6, 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 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

()
'(1 2 3)

[1 2 3]
[]
```

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.

For instance,

```
(+ 1 2)
```

4 Special forms; the `def` and `defn` 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

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)
```

5 The `quote`, `'`

6 Conditional forms

7 `do`, for sequential computation