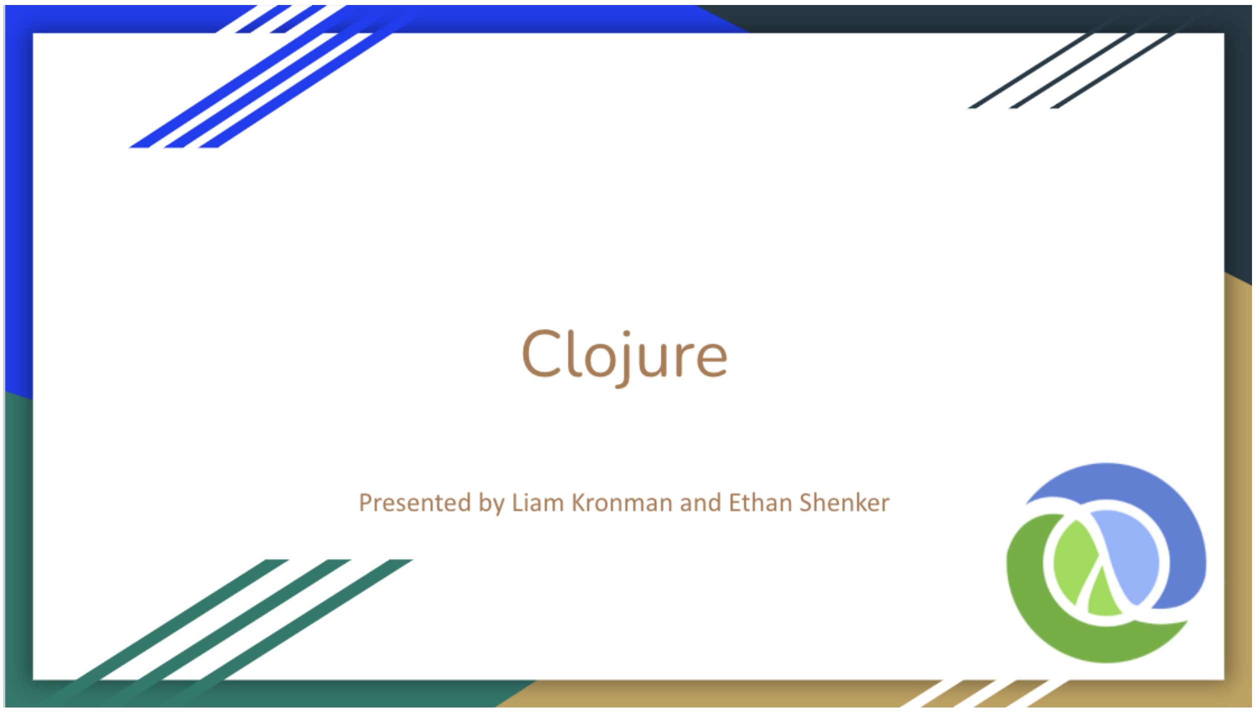
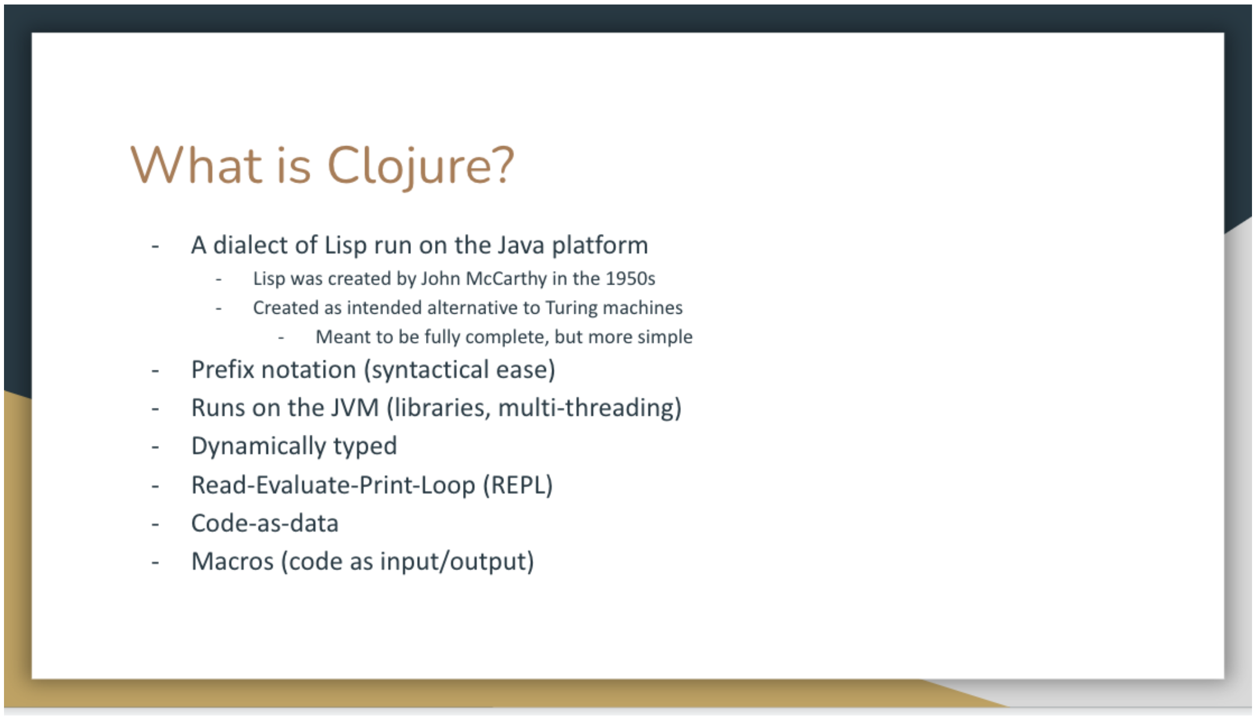
P2: TEDxSoftDevVIII - Clojure

Team Etham: Liam Kronman & Ethan Shenker



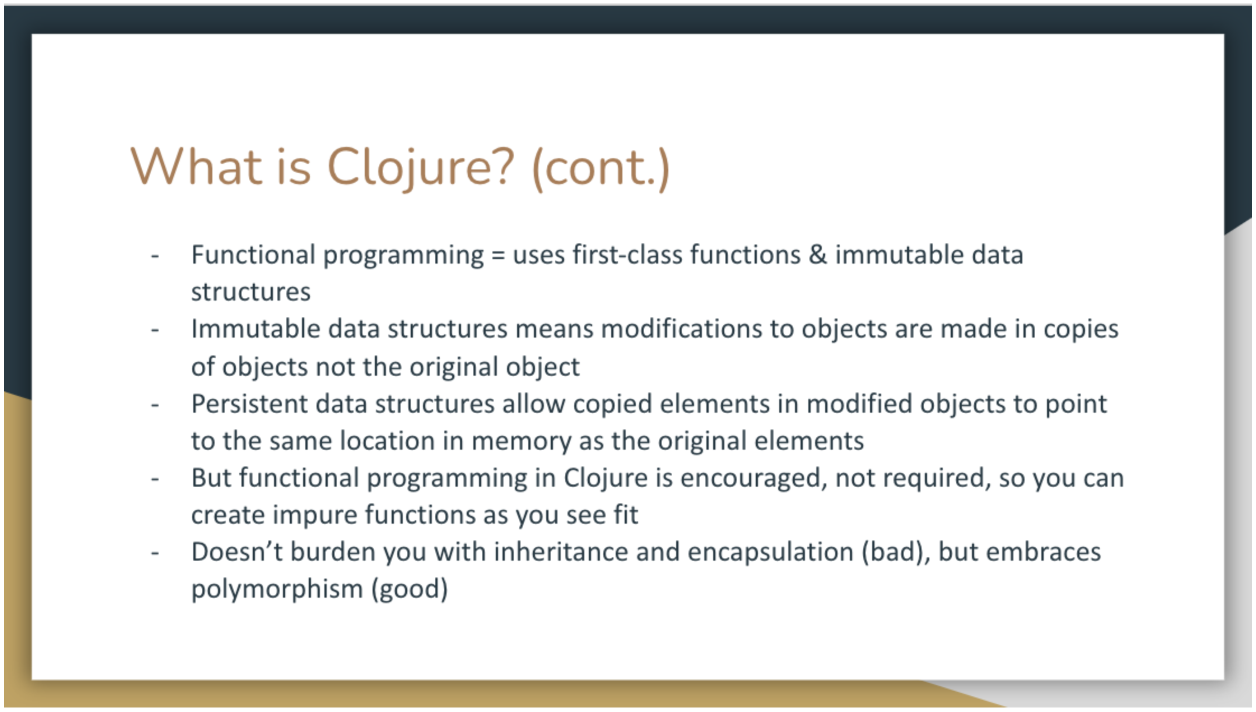
Clojure

Presented by Liam Kronman and Ethan Shenker



What is Clojure?

- A dialect of Lisp run on the Java platform
  - Lisp was created by John McCarthy in the 1950s
  - Created as intended alternative to Turing machines
    - Meant to be fully complete, but more simple
- Prefix notation (syntactical ease)
- Runs on the JVM (libraries, multi-threading)
- Dynamically typed
- Read-Evaluate-Print-Loop (REPL)
- Code-as-data
- Macros (code as input/output)

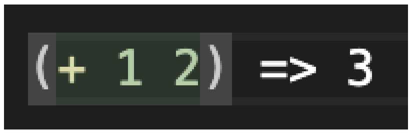


What is Clojure? (cont.)

- Functional programming = uses first-class functions & immutable data structures
- Immutable data structures means modifications to objects are made in copies of objects not the original object
- Persistent data structures allow copied elements in modified objects to point to the same location in memory as the original elements
- But functional programming in Clojure is encouraged, not required, so you can create impure functions as you see fit
- Doesn't burden you with inheritance and encapsulation (bad), but embraces polymorphism (good)

But what is Clojure... really?

Q: What is the REPL?
A: Standalone feature for Clojure, allows for interactive, real-time development of your code (live!)
Example:



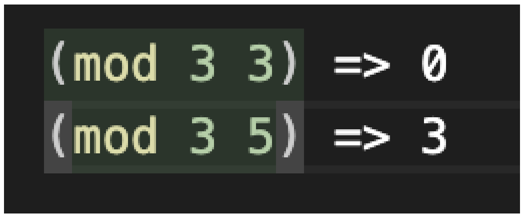
```
(+ 1 2) => 3
```

The above is a snippet of Clojure code from Visual Studio Code. The arrow and 3 are the result of evaluating the lefthand expression through REPL.

Suppose you'd like to use Clojure for the following Project Euler problem:
If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
Find the sum of all the multiples of 3 or 5 below 1000.

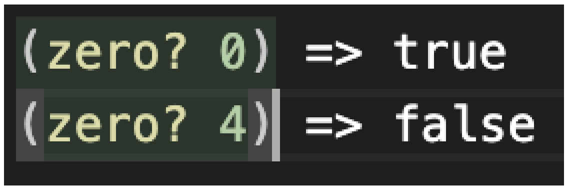
Clojure promotes decomposition, testing each function at each step before completing a larger function
To solve this Project Euler problem using Clojure:

1. Understand and evaluate modulo functions (finds the remainder when dividing two numbers)



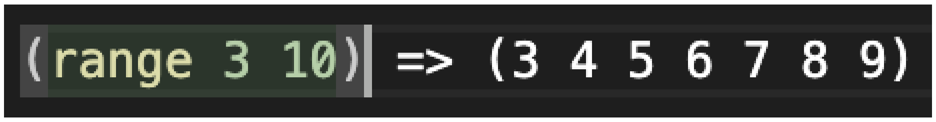
```
(mod 3 3) => 0
(mod 3 5) => 3
```

2. Understand and test zero? function (returns boolean of whether value is 0)



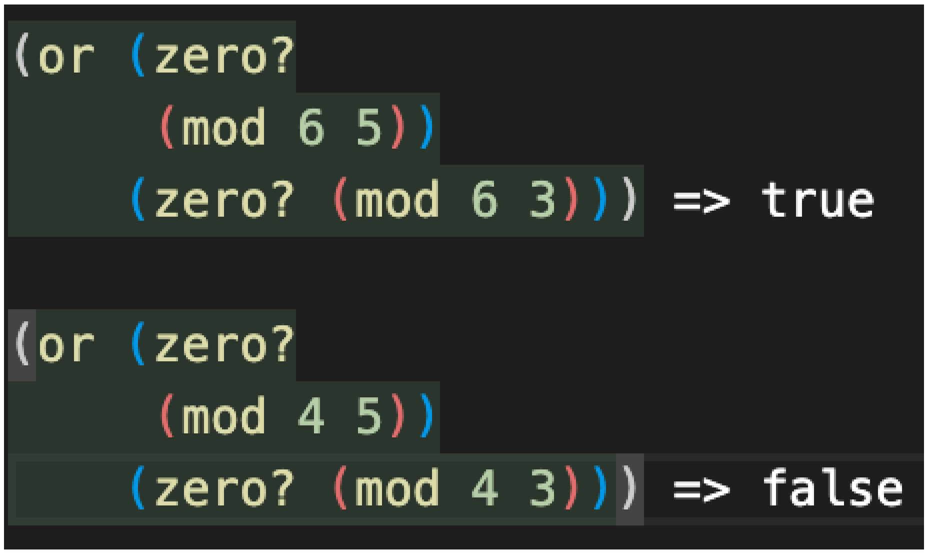
```
(zero? 0) => true
(zero? 4) => false
```

3. Understand and test range function (creates a list of values between two numbers)



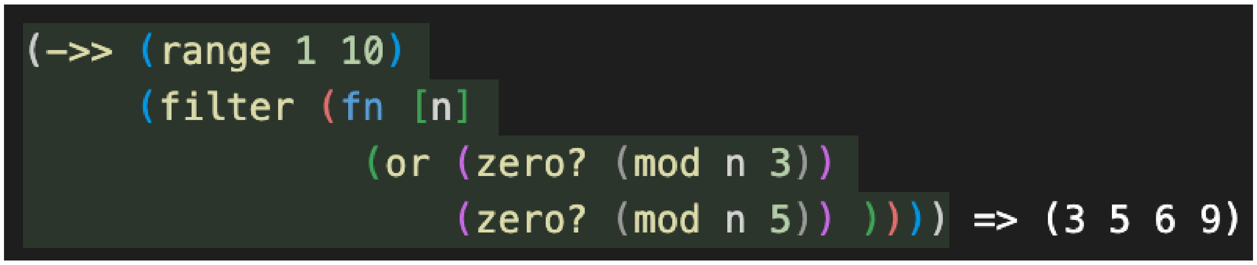
```
(range 3 10) => (3 4 5 6 7 8 9)
```

4. Put functions together to test on a smaller (more manual) scale (determine whether value is divisible by 5 or 3)



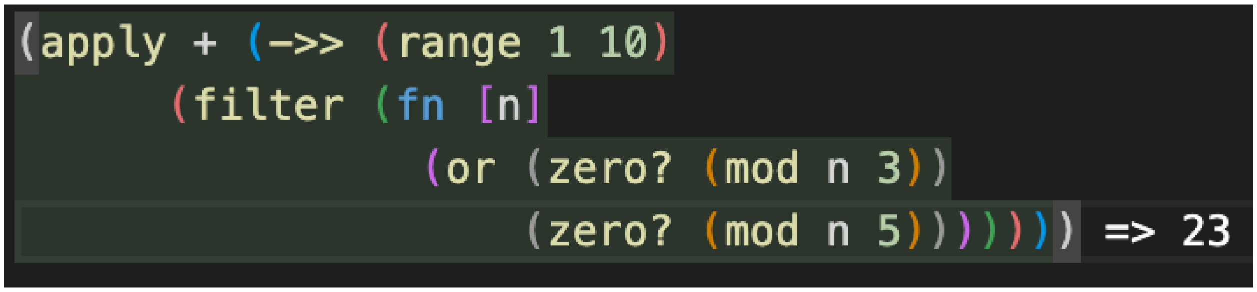
```
(or (zero?
     (mod 6 5))
    (zero? (mod 6 3))) => true

(or (zero?
     (mod 4 5))
    (zero? (mod 4 3))) => false
```

5. Create function to solve smaller version of problem (find natural numbers under 10 divisible by 3 or 5)



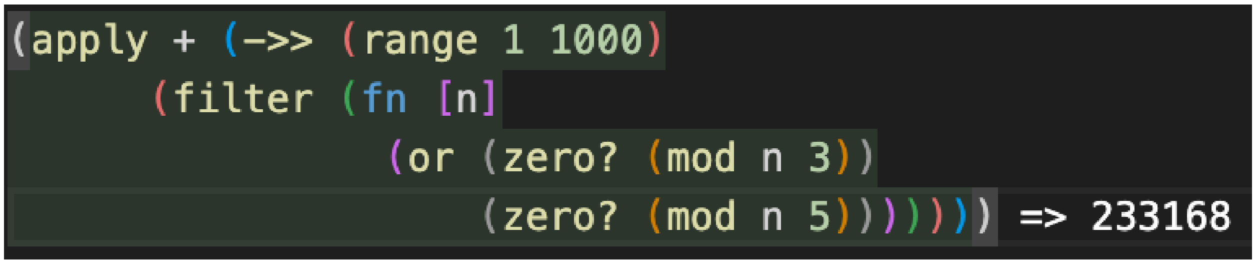
```
(->> (range 1 10)
     (filter (fn [n]
               (or (zero? (mod n 3))
                   (zero? (mod n 5)) )))) => (3 5 6 9)
```

6. Add the apply function (applies a certain operation to all elements in a list)

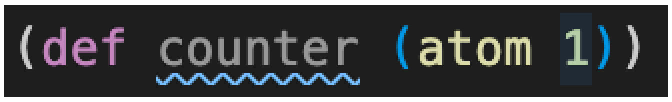


```
(apply + (->> (range 1 10)
      (filter (fn [n]
                (or (zero? (mod n 3))
                    (zero? (mod n 5))))))) => 23
```

7. Change range from 10 to 1000

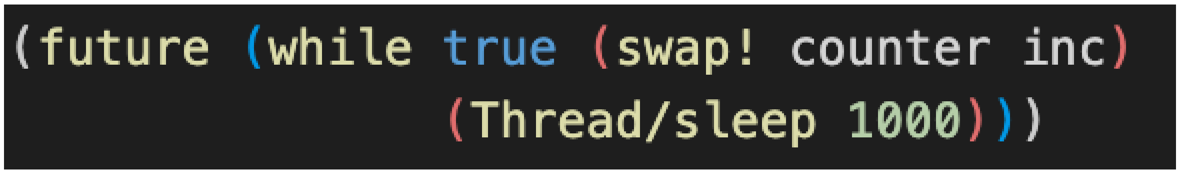


```
(apply + (->> (range 1 1000)
      (filter (fn [n]
                (or (zero? (mod n 3))
                    (zero? (mod n 5))))))) => 233168
```



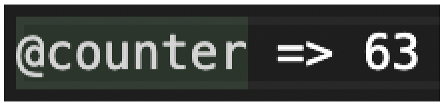
```
(def counter (atom 1))
```

Counter has been defined to be the atom value 1, meaning it will not be treated as an immutable data structure



```
(future (while true (swap! counter inc)
                    (Thread/sleep 1000)))
```

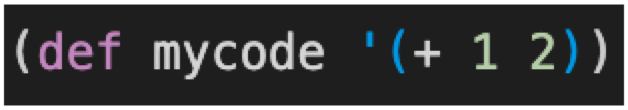
Future creates another thread, allows us to run other code in the background while we continue to program. Example of parallel processing.



```
@counter => 63
```

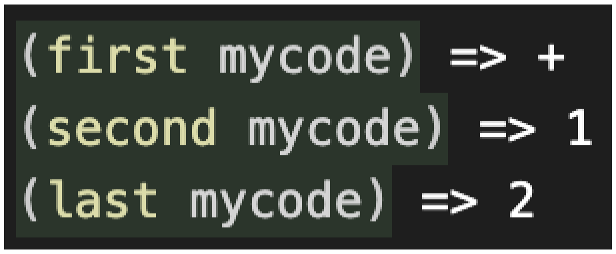
@ is the deref function

Code-as-data example



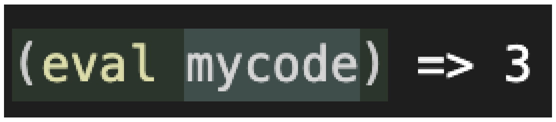
```
(def mycode '(+ 1 2))
```

‘ tells Clojure not to evaluate the function that follows but preserve it as a list



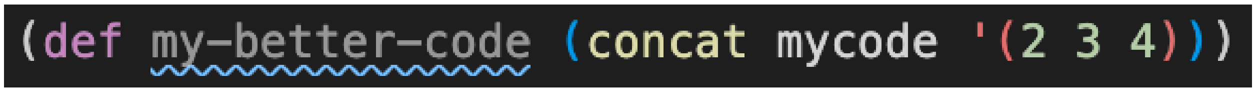
```
(first mycode) => +
(second mycode) => 1
(last mycode) => 2
```

first, second, and last are functions that reference indices of a list



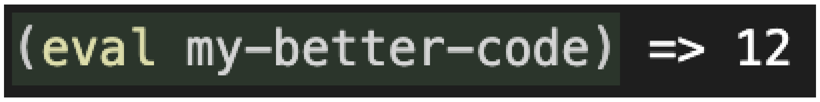
```
(eval mycode) => 3
```

eval evaluates mycode as a function, not a list



```
(def my-better-code (concat mycode '(2 3 4)))
```

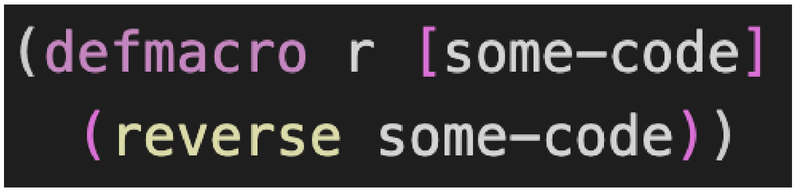
my-better-code is a function that concatenates 2, 3, and 4 to the function mycode



```
(eval my-better-code) => 12
```

Evaluates mycode with the new values added

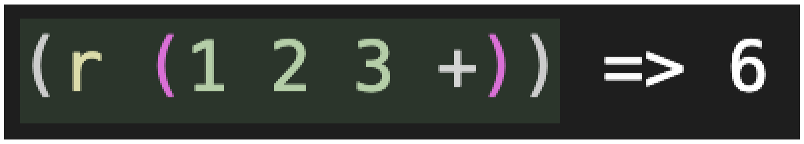
In Clojure, we can utilize Macros, which allow us to write code that can effectively write other code.



```
(defmacro r [some-code]
  (reverse some-code))
```

Here, we use the defmacro keyword to (you guessed it) define a macro, which we name r. This macro takes in a piece of data (or some-code) as its argument, and reverses its contents using the reverse function.

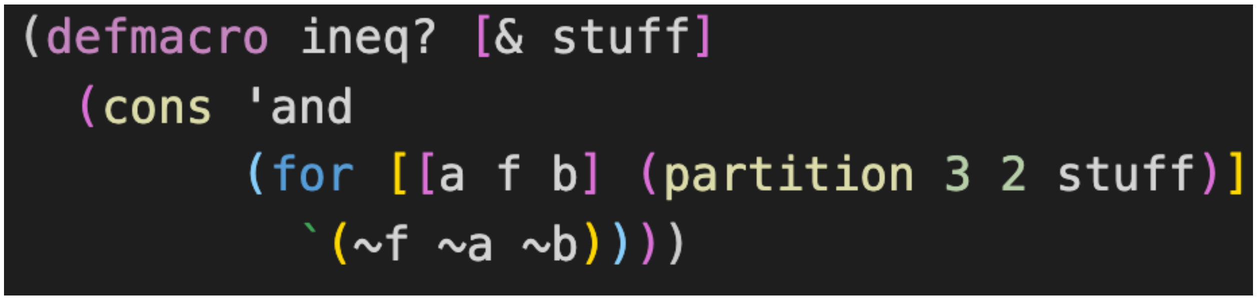
We can then use this macro that we've defined to manipulate data structures in new ways, such as the following:



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

, where our macro "r" reverses its input to then create an expression that Clojure's native syntax can understand.

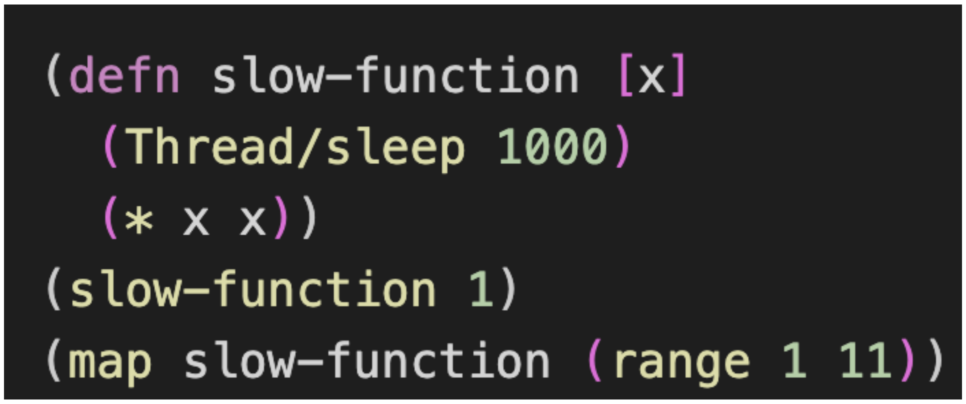
Macros can be used to do extremely complex operations, such as the partitioning of an unorganized input, which is what the following macro does:



```
(defmacro ineq? [& stuff]
  (cons 'and
        (for [[a f b] (partition 3 2 stuff)]
          `(~f ~a ~b))))
```

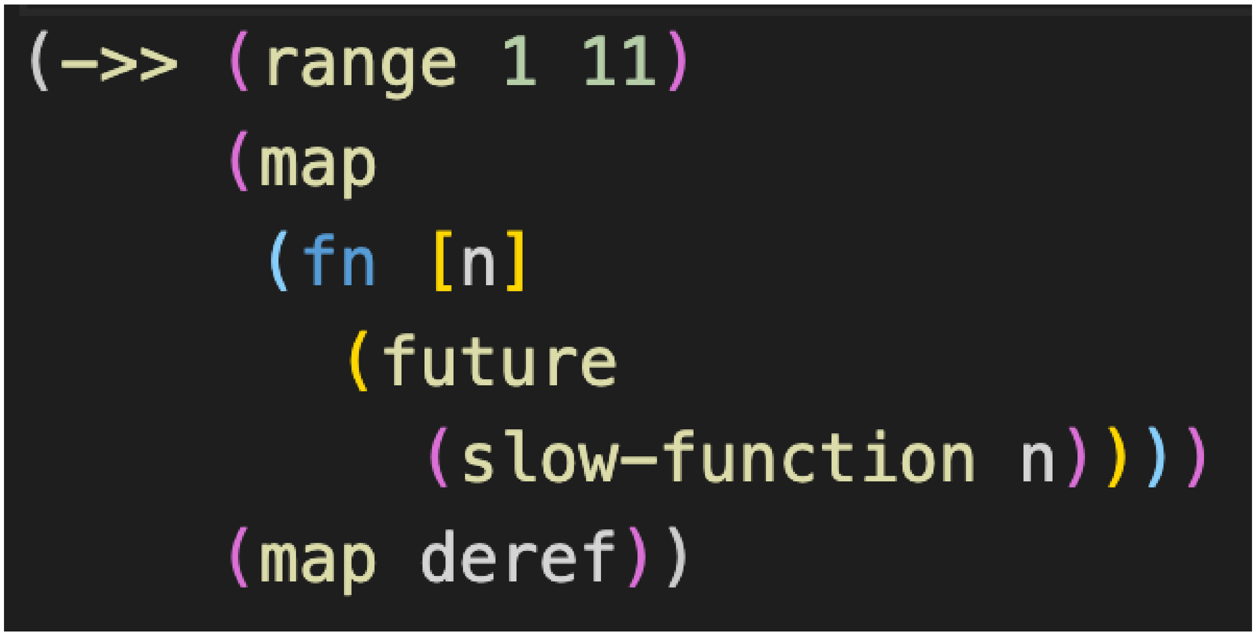
Here, we create a macro that can take in multiple data structures as an argument. Without explaining the specific details, this function iterates through every three elements of the original input, and breaks them up into separate expressions, where the elements are organized in a way such that this function can take a large in equality (1 < 3 < 5 < 7, for example), and translate it into prefix notation.

We also have the capability to use what's called multi-threading, in which a process is broken down into a series of subprocesses that are executed concurrently.



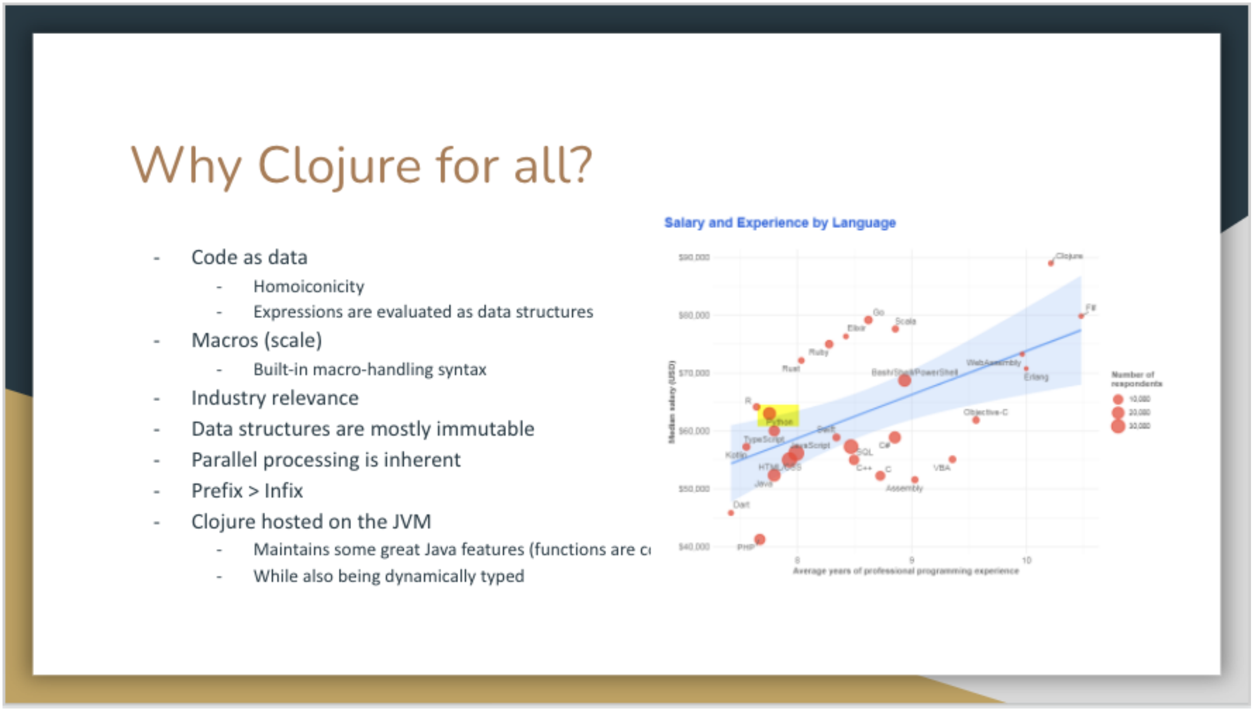
```clojure
(defn slow-function [x]
  (Thread/sleep 1000)
  (* x x))
(slow-function 1)
(map slow-function (range 1 11))
```

Here, we define a function that squares its argument after waiting for one second. We can use this at a small scale, but when we try to map this function on to a range of inputs, the execution time will be much slower, due to clojure's natural tendency to wait for one operation in a series to complete before proceeding to the next. We can utilize multi-threading to circumvent this issue.



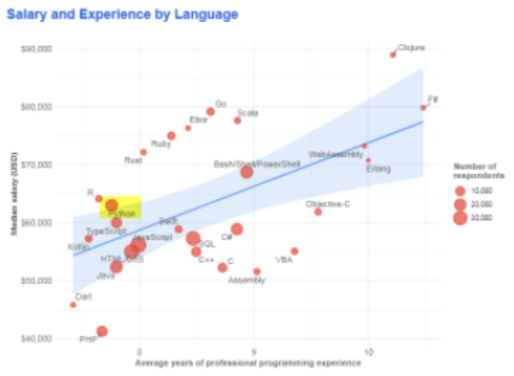
```clojure
(->> (range 1 11)
     (map
      (fn [n]
        (future
          (slow-function n))))
     (map deref))
```

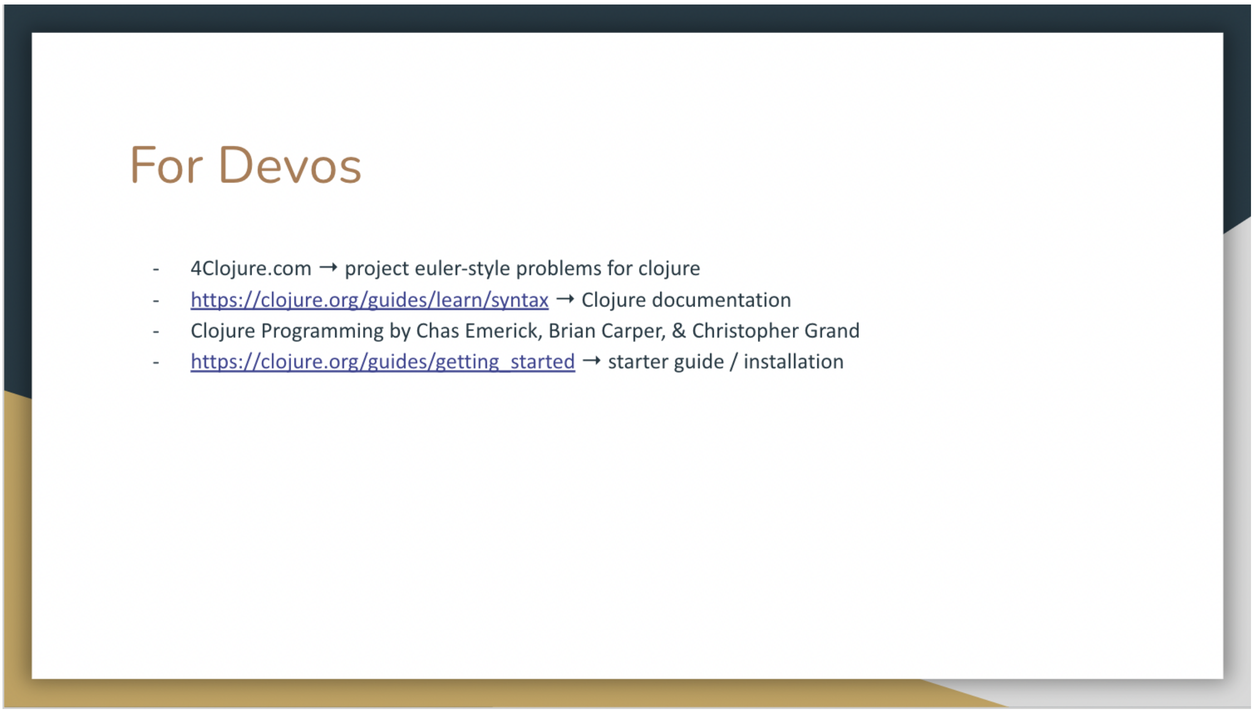
The future keyword creates a new thread for an operation to occur within, which means that this code fragment iterates through the range from 1 to 11, and creates a new thread in which our slow-function is ran with that argument. The runtime of this code segment is 1 second, rather than 10 seconds as exemplified in the fragment previous.



Why Clojure for all?

- Code as data
  - Homoiconicity
  - Expressions are evaluated as data structures
- Macros (scale)
  - Built-in macro-handling syntax
- Industry relevance
- Data structures are mostly immutable
- Parallel processing is inherent
- Prefix > Infix
- Clojure hosted on the JVM
  - Maintains some great Java features (functions are c
  - While also being dynamically typed





For Devos

- 4Clojure.com → project euler-style problems for clojure
- https://clojure.org/guides/learn/syntax → Clojure documentation
- Clojure Programming by Chas Emerick, Brian Carper, & Christopher Grand
- https://clojure.org/guides/getting_started → starter guide / installation

Here are the links again (if inaccessible):
4Clojure.com
https://clojure.org/guides/learn/syntax
https://clojure.org/guides/getting_started

Thank you!