CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 10 Clojure Introduction
Aug 29, 2022

What is Functional Programming

Elements of Functional Programming

Pure Functions

First Class Functions

Higher-Order Functions

Immutability

Lazy Evaluation

Recursion

Currying

Memoization

Destructuring

Collection Pipelines

List Compressions

Raw Data + functions

Raw Data + functions

```
class Person {
    private String firstName;
    private String lastName;
    private int age;
```

```
{:first-name “Roger”
 :last-name “Whitney”
 :age 21 }
```

filter (select), remove
map (fold)
reduce
transducers

Pure Functions

Functions with no side-effects

- Only depend on arguments
- Don't change state

```
class Foo {
    int bar

    public int notPure(int y) {
        return bar + y
    }

    public void alsoNotPure(int y) {
        bar = y
    }
```

Why important

Easier to
- debug
- test
- understand program

> OO makes code understandable by encapsulating moving parts.
>
> FP makes code understandable by minimizing moving parts.
>
> Michael Feathers

First Class Functions

Functions can be

- Assigned to variables
- Passed as arguments
- Returned from functions

Why important

- Flexibility
- Generality

Anonymous functions

- Lambdas
- Closures

Higher-Order Functions

Functions that operate on functions

Why important

Fewer details/
higher level logic

Concurrency

Immutability

Data structures can not be modified

Like Java's Strings

> OO makes code understandable by encapsulating moving parts.
>
> FP makes code understandable by minimizing moving parts.
>
> Michael Feathers

Why important

Concurrency

No need for private data

Easier to

- debug
- test
- understand program

Lazy Evaluation

Operations & functions evaluated

When used

Not when called

Why important

Simplifies logic

(def dice-rolls (map inc (repeatedly #(rand-int 6))))

(take 10 dice-rolls)

(2 5 5 4 6 6 3 4 2 5)

Type Checking

Strongly Typed

All type errors are detected

Rust is strongly typed at compile time - static type checking

Duck Typing

If a value can perform the operation it is the correct type

Clojure does duck typing

(+ a b)

Clojure

Developed by Rich Hickey

Started 2007

Variant of Lisp

Functional programming language

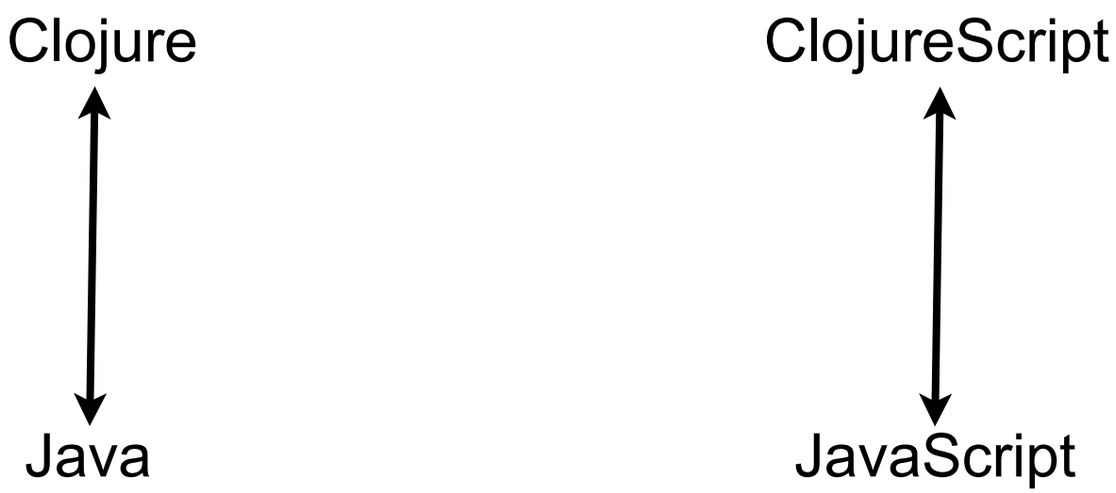
Dynamic typing

Interactive development - REPL

Tight Java Integration

Active development community

Variants



Base language the same

Few changes due differences between Java/Javascript

Development Environment

Visual Studio Code
  Calva plugins

IntelliJ
  Cursive plugin
  https://cursiveclojure.com

Command Line

Leiningen

Emacs
  CIDER

Vim
  Fireplace

Lots of Irritating Superfluous Parenthesis-LISP

reverse([1, 2, 3])

(reverse [1, 2, 3]) But not more than Java

But only () and they build up

```
(+ 5 (- 2 (/ 4 (* 2 (inc (read-string "123"))))))
```

Use editor that is parenthesis aware

Useful forms

- let
- ->

Resources

Clojure Home Page

http://clojure.org

Clojure Cookbook

Safari Books On-line

http://proquest.safaribooksonline.com.libproxy.sdsu.edu/

Elements of Clojure Code

symbols
keywords
literals
lists
vectors
maps
sets

functions
macros
special forms (functions)

REPL

Read-Eval-Print Loop

Executable code (program) in repl

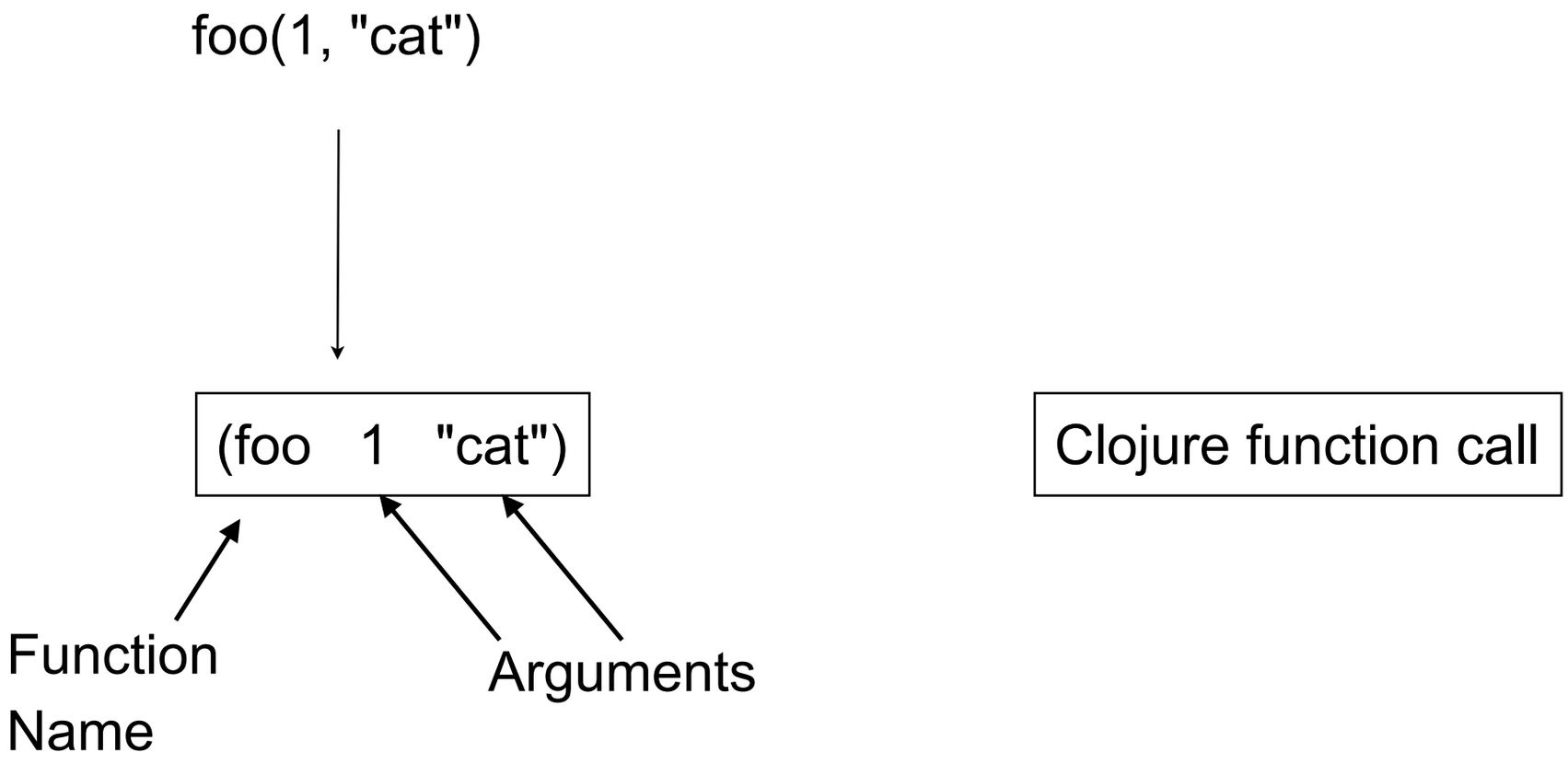
"hi there"

42

[1 2 3]

(+1 2)

Clojure Function Calls



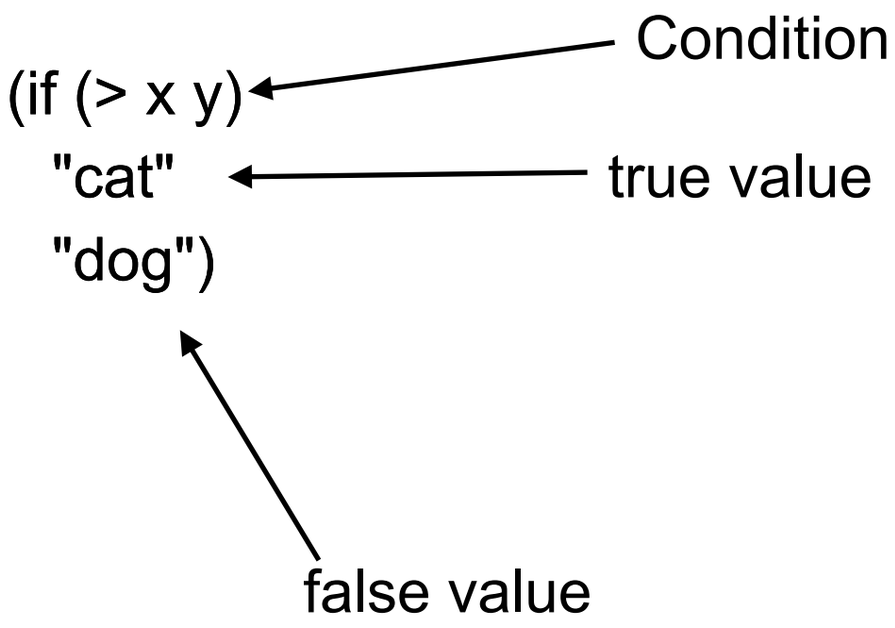
Some Basic Operations

Function	Result
(+ 1 2)	3
(+ 1 2 4 6)	13
(= "cat" "dog")	false
(= 1 1)	true
(= 1 1 2)	false
(even? 8)	true
(/ 10 2)	5
(/ 10 2 3)	5/3
(bit-shift-left 4 1)	8

Operators

No built-in operators

Just functions



Assignment

No built-in operators

Just functions

```
(def a 10)
(def b  (+ a 12))
(def a 20)
```

Called a binding which is sort of like assignment

No Precedence

a - b * c + d

(- a (+ (* b c) d))

Clojure expressions read inside out

Will see several ways to change this

Recursion
Higher Order Functions
The Functional Way

Vectors

Expandable, indexed list	[4 "cat" \c]
Fast insert at end	[4, "cat", \c]
Expensive insert in front	[ ]
Fast indexed lookup	

Vectors

(vector 8 4 2)	[8 4 2]
(nth [:a :b :c] 2)	:c
(first [1 2 3])	1
(second [1 2 3])	2
(third [1 2 3])	Error
(last [1 2 3])	3
(rest [1 2 3])	(2 3)

Compute the Sum

```
public float sum(ArrayList<float> list) {
   float sum = 0;
   for (int k = 0; k < list.length; k++)
      sum = sum + list.get(k);
   return sum;
}
```

Does not work in Functional World

No “for” statement

No side effects

Recursion replaces Iteration

```
(defn sum-1
  [list]
  (if (empty? list)
    0
    (+ (first list) (sum-1 (rest list)))))
```

(first list) returns first element in list

(rest list) returns list without the first element

(sum-1 [1 2 3])	6
(sum-1 (range 9900))	Stack over flow
(range 9900)	[0 1 2 3 4 5 ... 9898 9899]

Second Try

```
(defn sum-2
  [accumulator list]
  (if (empty? list)
    accumulator
    (sum-2 (+ accumulator (first list))
           (rest list))))
```

(sum-2 0 [1 2 3])	6
(sum-2 0 (range 9900))	Stack over flow

Recursive

(sum-1 [1 2 3])

(+ 1 (sum-1 [2 3]))

(+ 1 (+ 2 (sum-1 [3])))

(+ 1 (+ 2 (+ 3 (sum-1 []))))

(+ 1 (+ 2 (+ 3 0)))

(+ 1 (+ 2 3))

(+ 1 5)

6

(sum-2 0 [1 2 3])

(sum-2 1 [2 3])

(sum-2 3 [3])

(sum-2 6 (sum-2 [])

6

Tail Recursion Optimization

In a recursive function implementing a iterative process

The compiler can optimize the recursion into iteration

But JVM does not support tail recursion optimization

recur

```
(defn sum-3
  [accumulator list]
  (if (empty? list)
    accumulator
    (recur (+ accumulator (first list))
           (rest list))))
```

Replace the recursive call with recur

recur will call the function

But Clojure will convert to iteration

```
(sum-3 0 [1 2 3])          6

(sum-3 0 (range 9900))     49000050

(sum-3 0 (range 100000))   4999950000
```

One Name, Multiple Implementations

```
(defn sum-4
  ([list]
   (sum-4 0 list))
  ([accumulator list]
   (if (empty? list)
    accumulator
    (recur (+ accumulator (first list))
           (rest list)))))
```

```
(sum-4  [1 2 3])                6
(sum-4  0 [1 2 3])              6
(sum-4 (range 100000))          4999950000
(sum-4 0 (range 100000))        4999950000
```

Major Points

Recursion replaces “for” loops

Accumulators

Tail recursion optimization (recur)

But this is not the way to implement sum

reduce

```
(reduce + [1 2 3 4 5])
```

What versus How

What

```
(reduce + [1 2 3 4 5])
```

Less typing

Fewer details

Less cognitive load

More general solution

Code can be optimized

How

```
public float sum(ArrayList<float> list) {
  float sum = 0;
  for (int k = 0; k < list.length; k++)
    sum = sum + list.get(k);
  return sum;
}
```

Higher Order Functions

Function that acts on functions

```
(reduce + [1 2 3 4 5])
```

Timing tests

Code	Time
(sum-3 0 (range 100000))	54450.6 msecs
(sum-4 0 (range 100000))	26.1 msecs
(reduce + (range 100000))	6.5 msecs

(def data (range 1000000))

Code	Time
(sum-4 data)	~55 msecs
(reduce + data)	~22.5 msecs

The Functional Way

Raw data	Rich set of powerful functions on data
vectors	map
maps (hash table)	map-indexed
sequences	filter
	reduce
	remove
	keep
	zipper
	drop-while
	take-while
	partition
	interpose
	split-at
	etc.

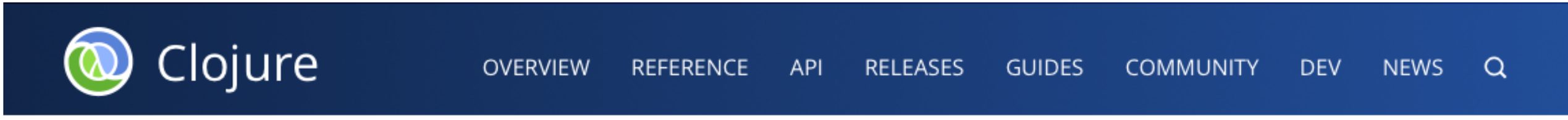
Immediate Goals

Recursion

Master use of built-in functions

Get comfortable with higher-order functions.

Clojure API



Cheatsheet

Clojure 1.11 Cheat Sheet (v54)

Download PDF version / Source repo

Many thanks to Steve Tayon for creating it and Andy Fingerhut for ongoing maintenance.

Documentation

clojure.repl/	doc find-doc apropos dir source pst javadoc (foo.bar/ is namespace for later syms)

Primitives

Numbers

Literals	Long: 7, hex 0xff, oct 017, base 2 2r1011, base 36 36rCRAZY BigInt: 7N

Relations (set of maps, each with same keys, aka rels)

Rel algebra	(clojure.set/) join select project union difference intersection index rename

Transients (clojure.org/reference/transients)

Create	transient persistent!
Change	conj! pop! assoc! dissoc! disj! Note: always use return value for later changes, never original!

Misc

https://4clojure.oxal.org/

4ever-clojure

home | github | Built with ❤ by @oxalorg and @borkdude

Keeping 4clojure alive forever! This website is completely static and evals code using sci. Suggestions / PRs welcome at github.com/oxalorg/4ever-clojure

Please note that 4ever-clojure is evaluated completely in the browser. So not all Java interop works, but some of it is the same in JS if you're lucky. Check cljs-cheatsheet for more info!

Problems (151)

No.	Name	Difficulty	Status
1	Nothing but the Truth	elementary	1/1 Passed!
2	Simple Math	elementary	1/1 Passed!
3	Strings	elementary	-
4	Lists	elementary	-
5	conj on lists	elementary	-