Clojure 101

Aula 0

Quem sou eu?

- Envolvido com computadores desde os 10 anos, primeiro computador aos 12 ,1996 (486 dx4 100Mhz - 8mb de RAM)
- Participei de cena hacker entre 2000 e 2004 (script kiddie)
- Fiz engenharia elétrica, desisti e fiz computação
- Participei de projetos com a Nokia (Symbian, Qt e Android)
- Ex desenvolvedor mobile e backend
- Mestre em Ciência da Computação, sou docente no Instituto Federal do Ceará
- Clojure desde 2013 - usando para pesquisa (análises)
- Full stack: VM -> OS -> NET- > SYS -> APPS -> SE

Quem são vocês?

- Nome e idade
- Background
- Nível de proficiência em Clojure
- Outros interesses

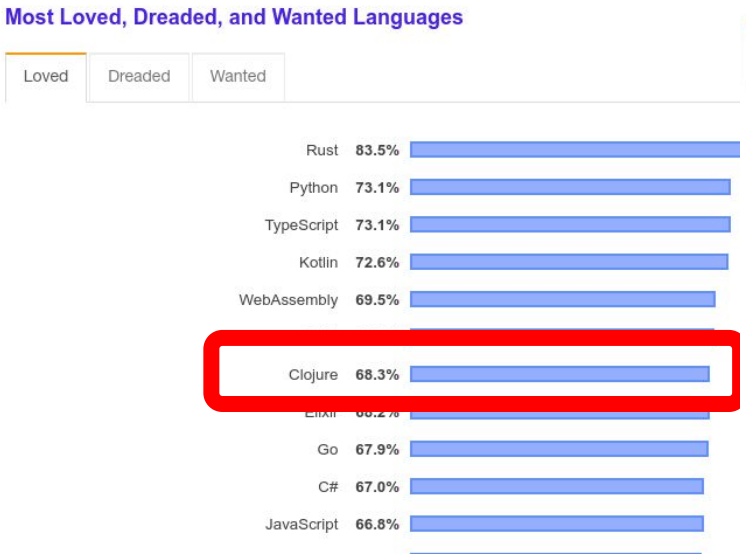
O que não vamos ver nesse curso

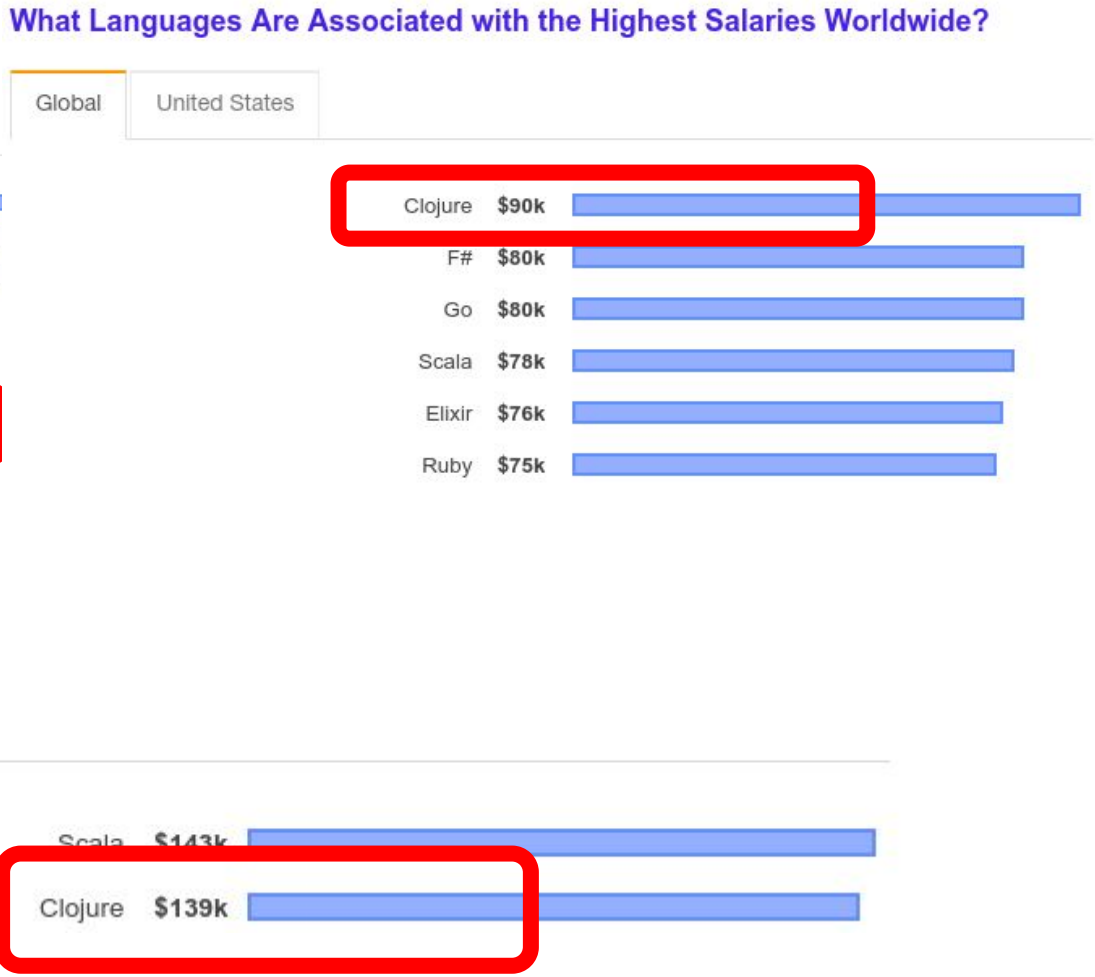
- Recursos avançados da linguagem (go routines, transdutores, etc...)
- Criação de Macros
- Desenvolvimento Web ;-)
- Manipulação de banco de dados

O que vamos ver nesse curso

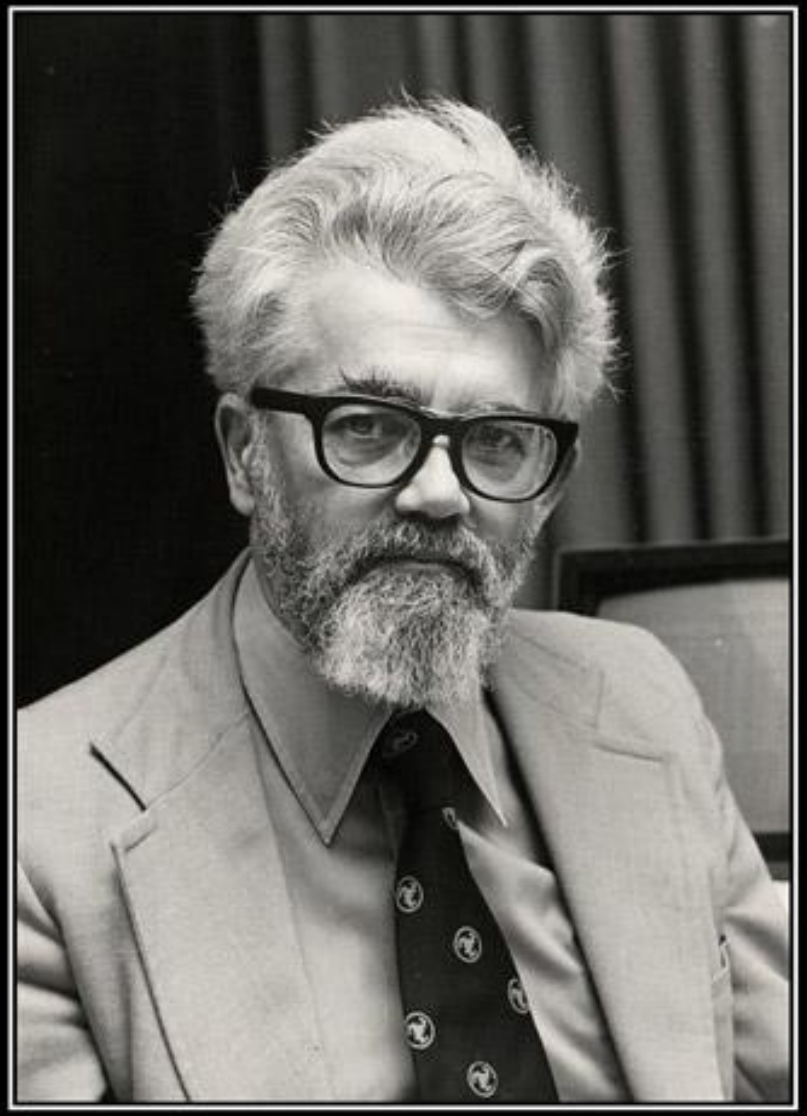
- Sintaxe da linguagem
- Como resolver problemas com recursão
- Strings
- Manipulação das estruturas de dados
- Manipulação de arquivos (csv e json)
- Um pouco de Java interop
- Como criar, rodar, testar e buildar projetos
- Algumas bibliotecas importantes
- Implementação de pequenos projetos
- Polimorfismo

Por que Clojure?





https://insights.stackoverflow.com/survey/2019#technology



PROGRAMMING

You're Doing It Completely Wrong.

Introdução

- LISP (LISt Processing) é uma linguagem de programação que foi concebida em 1958 por John McCarthy
- Ao lado de FORTRAN, pode ser considerada uma das primeiras linguagens de programação
- Inspirada fortemente pelo Cálculo Lambda (Church 1936)
- McCarthy também é o criador do Garbage Collector e da sintaxe do `if-then-else` que conhecemos hoje
- LISP também favorece a recursividade

Introdução

- LISP é homoicônica, ou seja, os programas são escritos na mesma estrutura de dados da linguagem (listas)
- LISP tem uma notação pré-fixada completamente parentizada
- Isso quer dizer que a expressão “2 + 5” deve ser escrita na forma “(+ 2 5)”
- LISP possui diversos dialetos, sendo os mais conhecidos: Common Lisp, emacs lisp, Scheme, Racket e Clojure
- Basicamente você escreve em AST

Introdução - Clojure

- Clojure é uma linguagem de programação dinâmica feita para a JVM (JS, .NET)
- Clojure é elegante
- Clojure herda as qualidades de LISP
- Clojure é uma linguagem funcional - as estruturas de dados são imutáveis e a maior parte de suas funções são puras
- Clojure simplifica a programação concorrente: atoms, refs, STM, CSP
- Interoperabilidade com JAVA é transparente (muitas libs)

```java
public static boolean isBlank(final CharSequence cs) {
    int strLen;
    if (cs == null || (strLen = cs.length()) == 0) {
        return true;
    }
    for (int i = 0; i < strLen; i++) {
        if (!Character.isWhitespace(cs.charAt(i))) {
            return false;
        }
    }
    return true;
}
```

https://commons.apache.org/proper/commons-lang/apidocs/src-html/org/apache/commons/lang3/StringUtils.html

```clojure
(defn blank?
  [s]
  (every? #(Character/isWhiteSpace %) s))
```

O que é Programação Funcional ?

- Estilo de programação baseado em duas teorias matemáticas
- Lambda Calculus:
  - Inventado em 1936 por Alonzo Church
  - Baseado em funções anônimas (lambdas)
  - Definições e aplicações
  - Sintaxe não-amigável
- Teoria das funções recursivas:
  - Stephen Kleene em 1938

O que é Programação Funcional (FP)?

- Baseado em definição e chamada de funções
- λ Calculus tem o mesmo poder computacional que a máquina de Turing (tese de Church-Turing)
- Em linguagens imperativas (LI) o mesmo nome (variável) pode ser associada a vários valores diferentes
- Em FP um nome é associado a apenas um valor
- Em LI novos valores podem ser associados ao mesmo nome através de comandos de repetição: i++ (por ex.)
- Em FP novos valores são associados com novos nomes através de chamadas recursivas

O que é Programação Funcional (FP)?

- Linguagens funcionais fornecem representações literais para suas estruturas de dados
- Funções como valores (**first class citizens**), ou seja, funções são tratadas como valores
- Funções podem ser atribuídas à variáveis, passadas como valor para outras funções(**higher order functions**) e podem ser retornadas

O que é Programação Funcional (FP)?

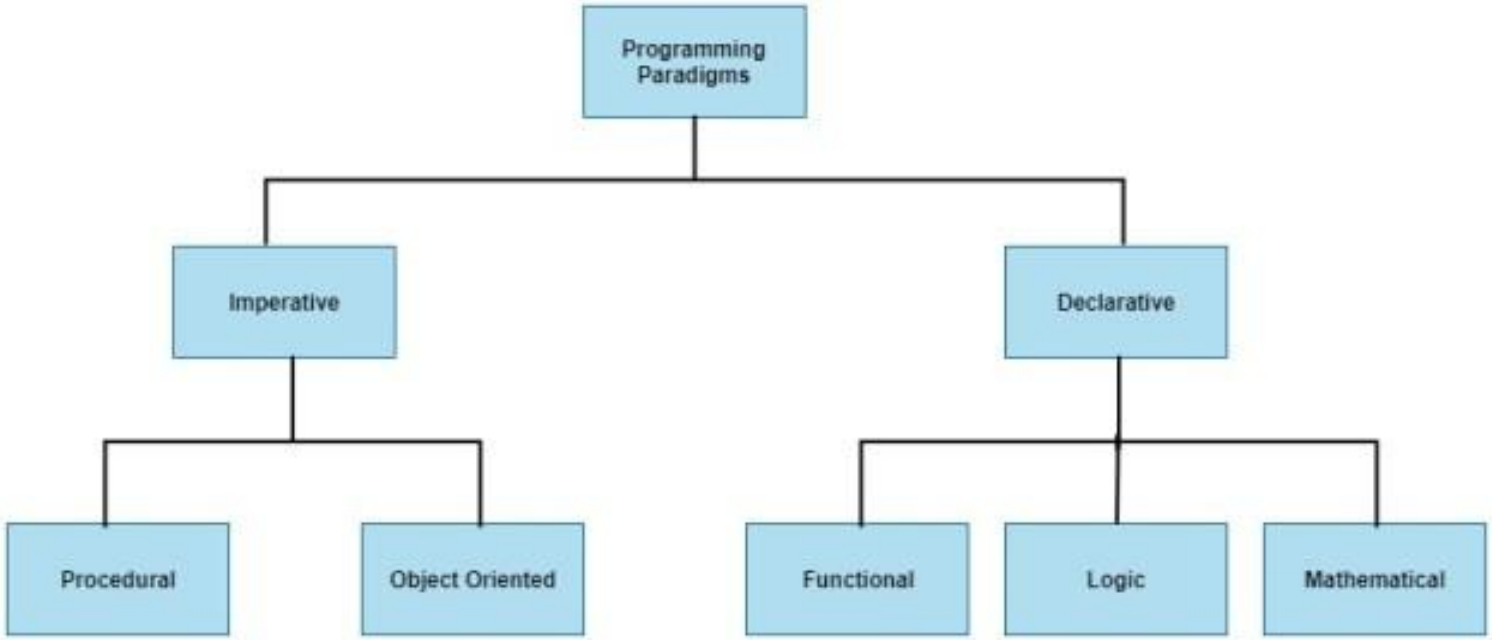
- Em Lambda Calculus tudo é função
- Números são funções (???)
- Booleanos são funções (???)
- Aritmética com funções
- Operadores booleanos como funções

Lambda Functions - Mary had a little lambda by Anjana Vakil

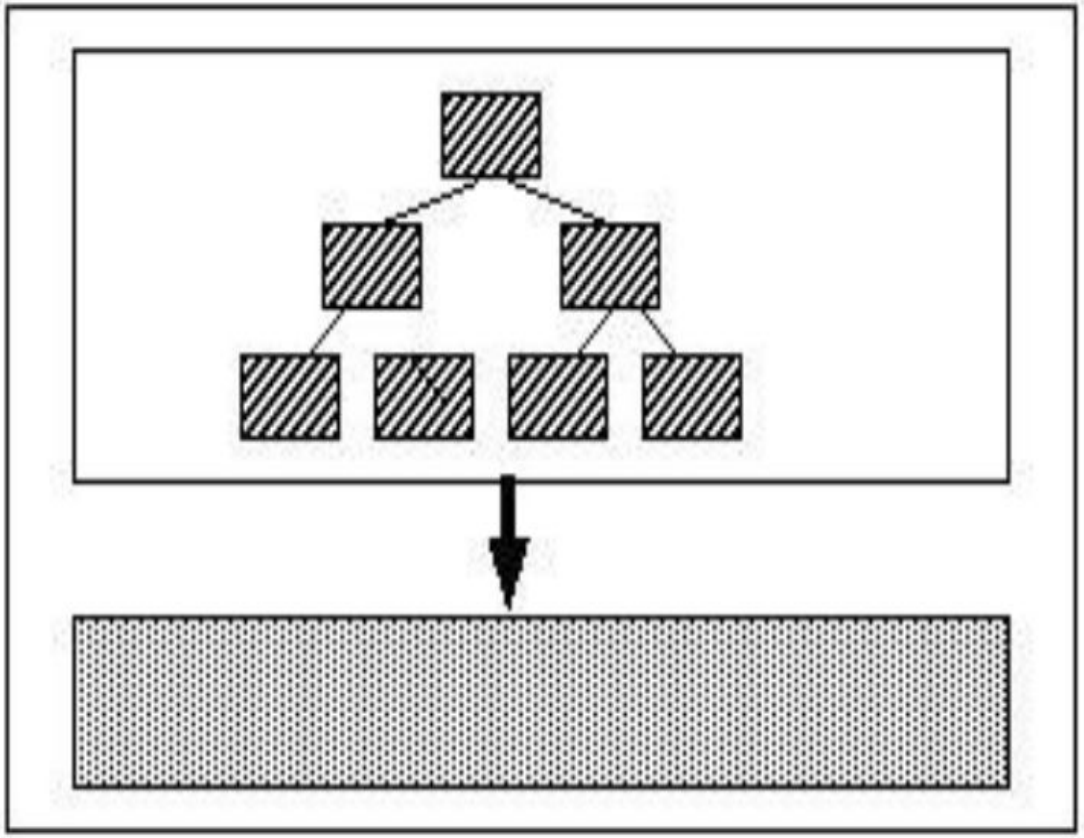
https://www.youtube.com/watch?v=9pmI-KK4dIA&t=937s

FP - outline

- Imutabilidade
- Funções puras* (sem side-effects)
- Funções de alta ordem
- Recursão para gerenciar estado
- Estruturas de Dados Persistentes
- Declaratividade:



Procedural Languages

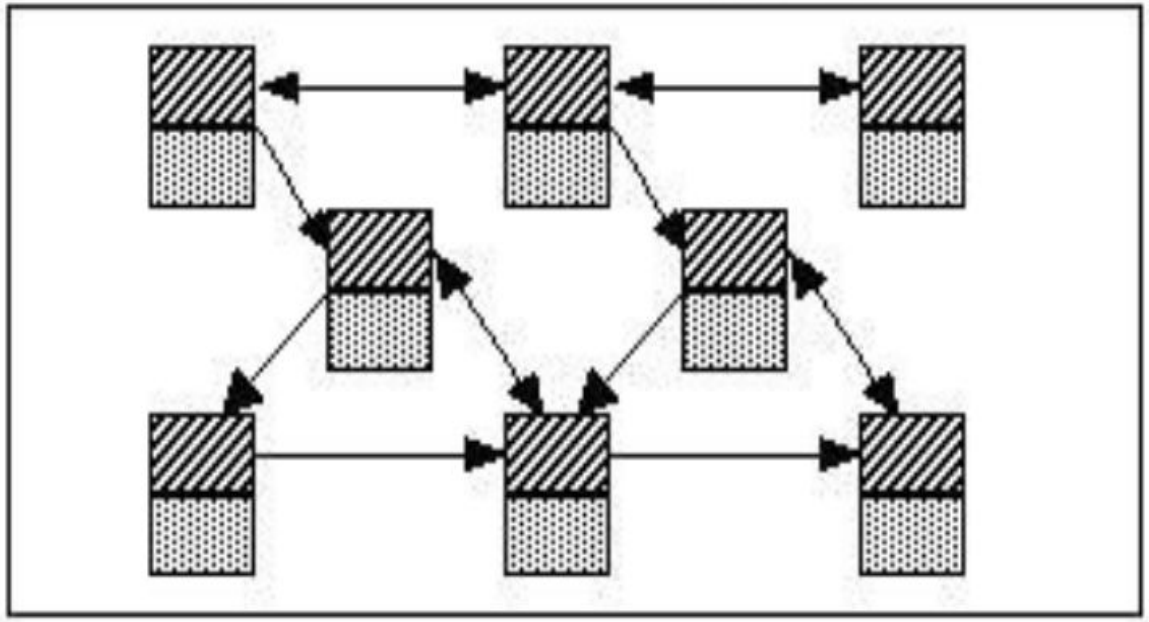


Computation involves code operating on Data

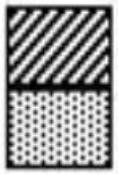
Code

Data

Object-Oriented Languages

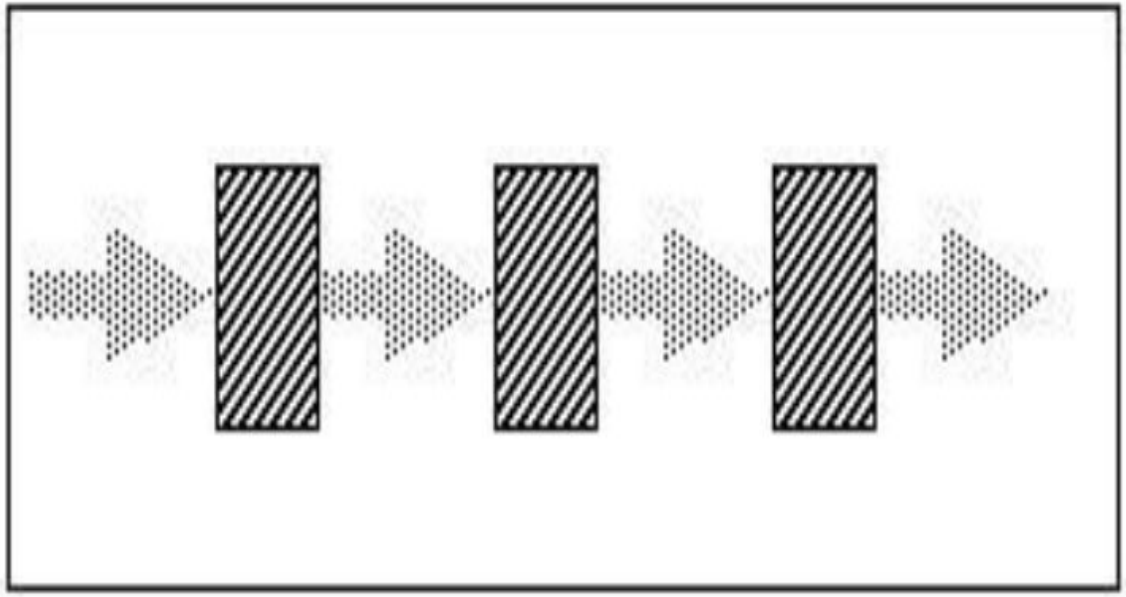


An object encapsulates both code and data

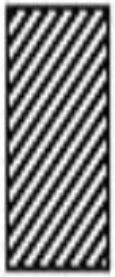


Computation involves objects interacting with each other

(Pure) Functional Languages



Data has no
independent existence

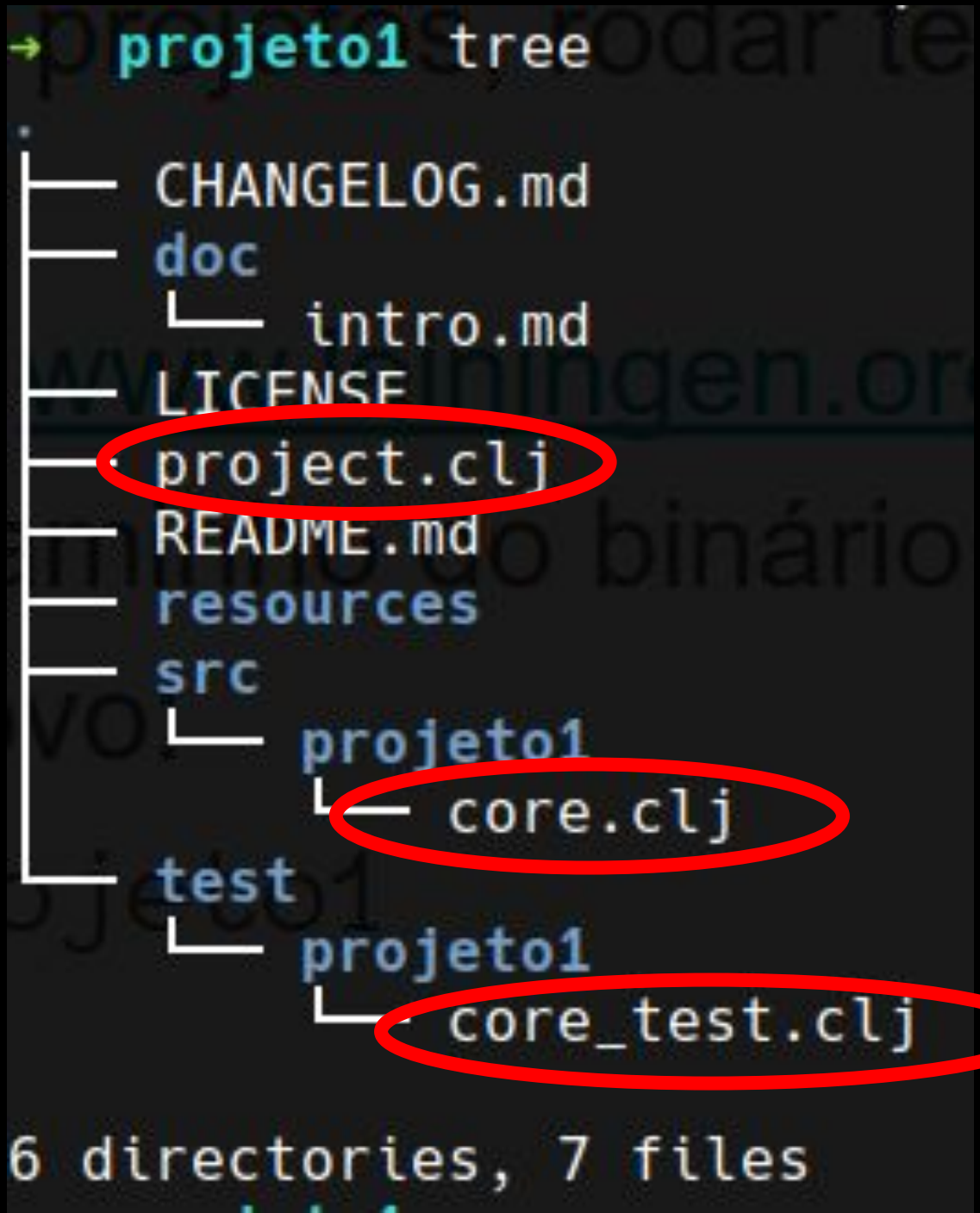


Code (Functions)

Computation involves data
flowing through functions

Leiningen

- Leiningen é a maneira mais simples de se usar Clojure
- Com ela você pode criar projetos, rodar testes, gerar executáveis (JAR)
- Baixar o leiningen em www.leiningen.org
- Colocar no PATH, o caminho do binário
- Criando um projeto novo:
  - **`lein new app projeto1`**
  - **`lein run`**
  - **`lein test`**
  - **`lein uberjar`**


→ projeto1 tree
.
├── CHANGELOG.md
├── doc
│   └── intro.md
├── LICENSE
├── project.clj
├── README.md
├── resources
├── src
│   └── projeto1
│   └── core.clj
└── test
    └── projeto1
        └── core_test.clj
6 directories, 7 files

Hands On

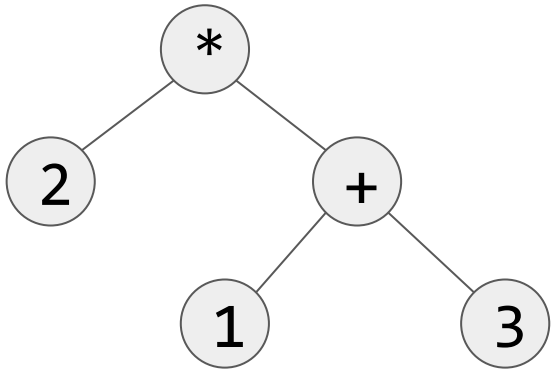
lein repl

Tipos primitivos

- Numéricos:
  - Long: `1, 2, 1000`
  - Double: `3.14`
  - Ratio: `1/4`
  - Binário, Octal,...: `2r1010, 8r172`
- Char e Strings:
  - Character: `\a, \b, \c, \d`, ...
  - String: `“blah blah blah”`
- Keyword: `:key, :birthdate, :name, :salary`
- Boolean: `true, false`
- Nulo: `nil`

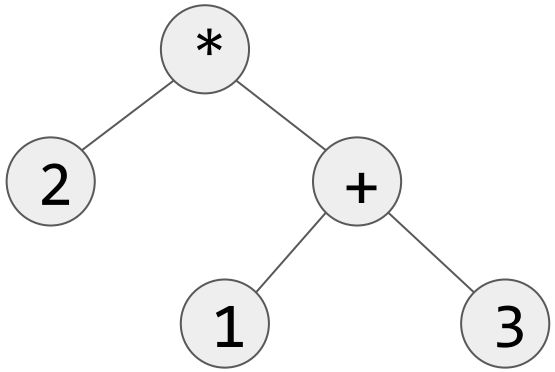
“Operadores” aritméticos (funções)

- Soma:
  - `(+ 1 2), (+ 1 2 3), (+ x y),...`
  - Aceita zero ou mais parâmetros
  - Se chamada sem parâmetros, **retorna a identidade** da operação (**zero**)
- Subtração:
  - Semelhante à soma, porém não possui identidade
- Multiplicação:
  - `(* 2 3) => 6`
  - `(* 2 (+ 1 3)) => 8`
- Divisão:
  - `(/ 12 4) => 3`
  - `(/ 1 4) => 1/4`



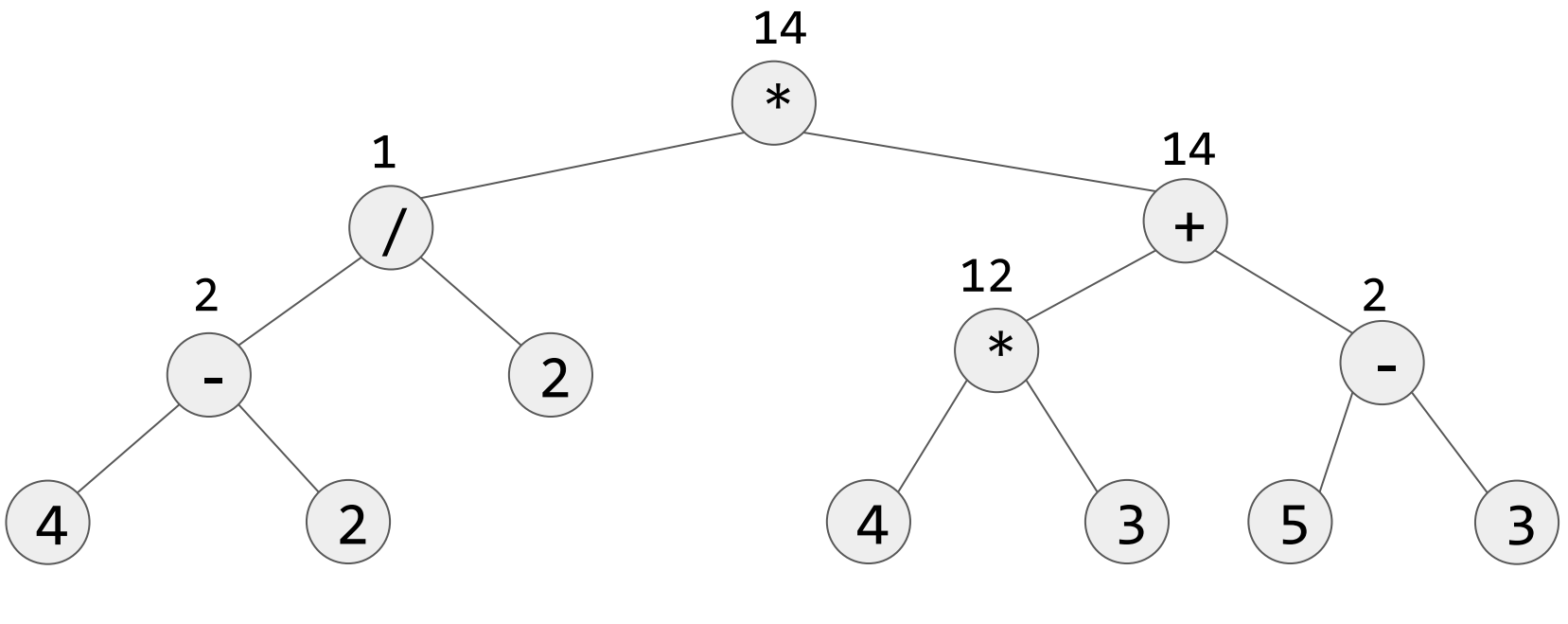
Converta as expressões abaixo para a notação de clojure

- `2 + 3 => (+ 2 3)`
- `4 + 3 * 5`
- `((4 - 2) / 2) * (4 * 3 + (5 - 3))`
  - `14`
- `1 + 2 + 3 + 4 + 5`
- `5 * 4 * 3 * 2 * 1`
- `(20 / 4) * (14 - 33)`
  - `-95`
- `666 - 123 * 90 - 1`
  - `-10405`



Árvore Sintática Concreta

- `((4 - 2) / 2) * (4 * 3 + (5 - 3))`



- `(* (/ (- 4 2) 2) (+ (* 4 3) (- 5 3)))`

Outras funções

- Divisão inteira:
  - `=> (quot 10 3)`

    `3`
- Resto da divisão :
  - `=> (rem 5 2)`

    `1`

Definindo variáveis

- Como sabemos, variáveis em FP não mudam
- Em clojure existe uma sintaxe para definir novas variáveis:

```
=> (def nome-da-variável expressão)
=> (def idade 10)
=> (def mensagem “Olá, Mundo!”)
=> (def valor 50.5)
=> (def pi 3.14)
=> (def raio 14.5)
=> (def area (* pi (* raio raio)))
```

Definindo nossa primeira função

- Funções **SEMPRE DEVEM RETORNAR VALORES**
- Função é o elemento principal da linguagem:

Sintaxe:

```
(defn nome-da-função
  “Documentação opcional”
  [param1 param2 & params]
  corpo-da-função)
```

```
=> (defn mensagem []
     “Retorna mensagem”
     “Olá, Mundo!”)
=> (defn mensagem-personalizada
     [nome]
     (str “Olá ” nome)
```

Chamando suas funções

```
Sintaxe:
=> (mensagem)
“Olá, Mundo!”
nil

=> (doc mensagem)
user/mensagem
([])
  Documentação Opcional

=> (println (mensagem-personalizada “Vladymir”))
“Olá, Vladymir”
```

Defina as seguintes funções

- **sucessor**: recebe um número e retorna o número mais um
- **antecessor**: recebe um número e retorna o número menos um
- **soma**: recebe dois números e retorna sua soma
- **sub**: recebe dois números e retorna sua subtração
- **mult**: multiplica dois números
- **div**: divide dois números*
- **area-circ:** recebe um raio e retorna a área do círculo

Operadores Relacionais (comparação)

- Igualdade:
  - `(= 1 1) => true`
  - `(= 2 (+ 1 1)) => ?`
  - `(= “a” “b”) => false`
  - `(not= 1 1 1) => false`
  - `(not= “a” “b” “c”) => true`
- Maior, Menor ou igual:
  - `(< 4 3) => false`
  - `(< 3 5) => true`
  - `(>= 4 4) => true`
  - `(> 10 6 5 4 3 2 1) => true`
  - `(<= 10 9) => false`

Defina as seguintes funções

- **maior?**: recebe uma idade e retorna true se a idade for maior ou igual a 18
- **menor?**: recebe uma idade e retorna true se a idade for menor que 18 (use a função anterior)

- **baixo?**: recebe uma altura em metros e retorna true se a altura for abaixo de 1.6
- **alto?**: recebe uma altura em metros e retorna true se for maior ou igual a 1.8

- **par?**: recebe um número e retorna **true** se o número for par, false caso contrário
- **impar?**: o contrário da função par

Operadores Lógicos

- and, or, not:
  - `(and true true) => true`
  - `(and 1 2) => ?`
  - `(and false true) => false`
  - `(or false true) => true`
  - `(or 1 2) => ?`
  - `(not false) => true`
  - `(not true) => false`

Expressões Condicionais

Sintaxe:

```
(if expr
   then
   else)
```

Sempre retorna um valor (then ou else)

nil sempre será avaliado como false

- **Se seu if possuir somente um branch, use when**

```
(when expr
   then)
```

- **Cada branch só permite uma expressão, caso necessite executar duas ou mais exprs use do:**

```
(do
   expr1
   expr2
   ...)
```

Expressões Condicionais

Para construir um triângulo é necessário que a medida de qualquer um dos lados seja menor que a soma das medidas dos outros dois e maior que o valor absoluto da diferença entre essas medidas.

$$|b-c| < a < b+c$$
$$|a-c| < b < a+c$$
$$|a-b| < c < a+b$$

No projeto aula0, no arquivo core.clj, complete a função **eh-triangulo?** para verificar se os três lados recebidos como parâmetro formam um triângulo. Escreva mais alguns testes no arquivo core-test.clj

Obs. escreva sua própria função absolute |x|

Expressões Condicionais

A Invillia resolveu dar um aumento de salário aos seus funcionários =). Você foi contratado para desenvolver o programa que calculará os reajustes.

- Complete a função **novo-salario** que recebe o salário de um funcionário e retorne o novo salário com reajuste segundo o seguinte critério, baseado no salário atual:
- salários até R$ 280,00 (incluindo) : aumento de 20%
- salários entre R$ 280,00 e R$ 700,00 (incluindo): aumento de 15%
- salários entre R$ 700,00 e R$ 1500,00 (incluindo): aumento de 10%
- salários entre R$ 1500,00 e R$ 3000 (incluindo): aumento de 5%
- salários acima de R$ 3000 não devem receber aumento

Execute os testes com **lein test**

Expressões Condicionais - cond

Sintaxe:

```
(cond
   expr1
   then1

   expr2
   then2

   expr3
   then3

   ...

   :else
   else
   )
```

Refatore as funções **eh-triangulo** e **novo-salario** para usar **cond** ao invés de **if**

Expressões let

Sintaxe:

```
(let [v1 expr1]
  *do-something-with-v1*)
```

v1 não existe fora do escopo do let

Ex.:

```
(let [raio 5.3]
  (area-circ raio))
(let [raio 10
      area (area-circ raio)
      nova-area (- area 10)]
  (println nova-area))
```

Expressões let - Propriedades

- **let** permite você nomear suas “variáveis” como desejar (variable shadowing)
- **let** permite você usar as variáveis definidas anteriormente (kinda identity monad)
- **let** permite *destructuring* (veremos mais adiante)
- **let** permite uma ou mais expressões no seu corpo, essas expressões são avaliadas em ordem
- variáveis definidas no **let** desaparecem fora do escopo
- **Variantes: if-let, when-let**

Funções Recursivas

- São funções que **possuem chamadas a si próprias** dentro do seu corpo
- Usadas para substituir loops tradicionais (for e while)
- Também são usadas para modificar estados imutáveis
- Precisam de uma **condição de parada (caso base)**, usada para terminar a recursão, ou
- Podem não ter condição de parada em casos específicos (main loops, por ex.)
- **TCO**: caso especial onde a chamada recursiva está sempre na cauda (Tail Call Optimization)

Funções Recursivas - Exemplos

- Soma e multiplicação
- Cálculo de Fatorial
- Sequência de Fibonacci
- Potenciação: x^n
- Busca Binária
- **map, filter e reduce***

Qualquer loop (while ou for) pode ser substituído por uma função recursiva. Pode-se dizer que são análogas ao princípio da indução matemática.

Funções Recursivas - Fatorial

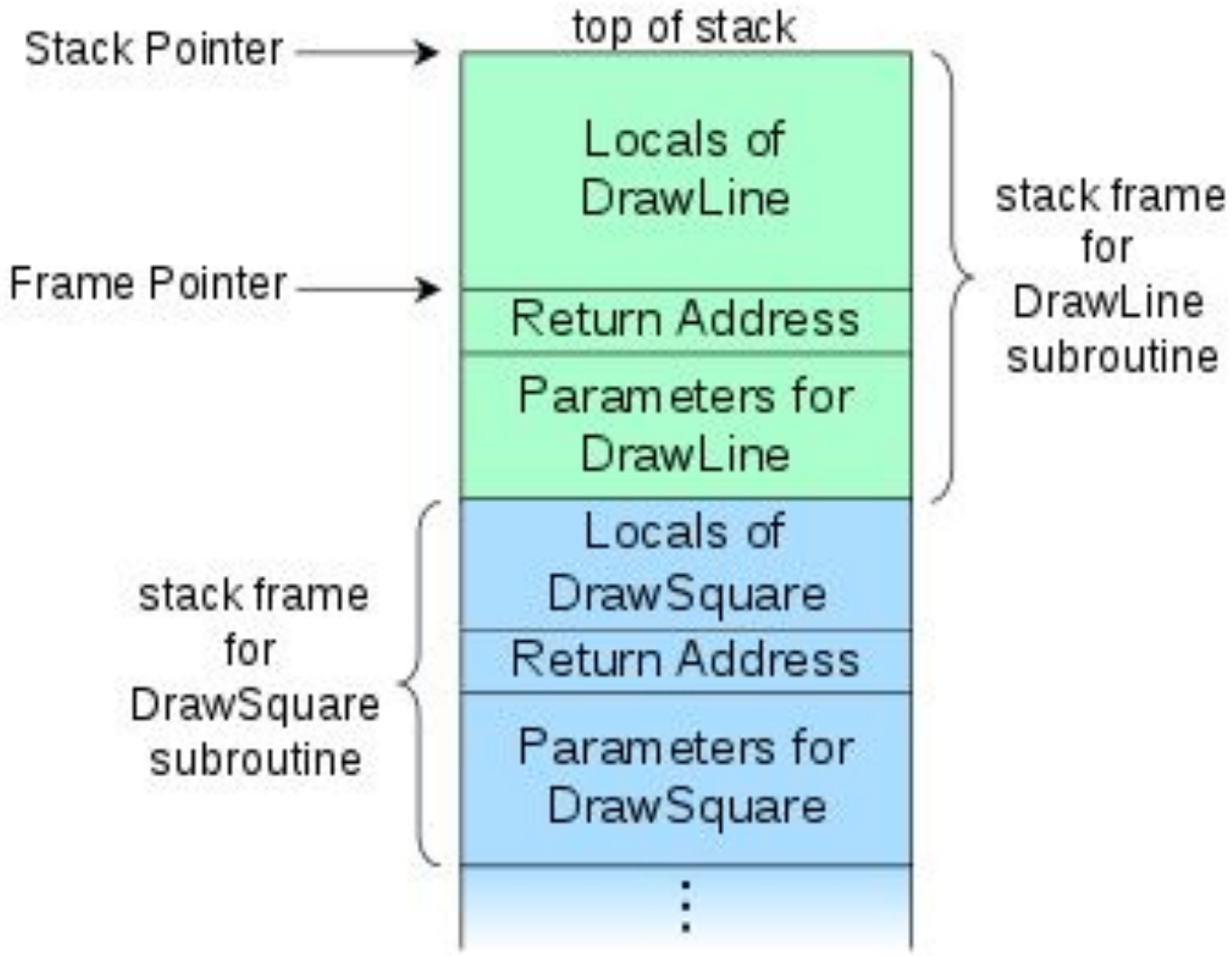
$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

```
fact(5) = 5 * fact(4)
          5 * 4 * fact(3)
          5 * 4 * 3 * fact(2)
          5 * 4 * 3 * 2 * fact(1)
          5 * 4 * 3 * 2 * 1 * fact(0)
          5 * 4 * 3 * 2 * 1 * 1
           5 * 4 * 3 * 2 * 1
          5 * 4 * 3 * 2
          5 * 4 * 6
          5 * 24
          120
```

Funções Recursivas - Considerações

- Internamente, a JVM (não só) armazena as chamadas de função em uma pilha
- Cada função é representada por uma *stack frame*, que armazena parâmetros, variáveis locais, **endereço de retorno**, etc…
- Cada chamada de procedimento cria um novo stack frame que é empilhado na pilha de chamadas (call stack)
- Quando a função retorna, o frame é removido da pilha, passando o controle para a função que a chamou
- Essa pilha tem um tamanho limitado e por isso não podemos abusar nas chamadas recursivas
- Como se resolve, afinal ?

Funções Recursivas - Considerações



Funções Recursivas - Fatorial em Clojure

```
(defn fatorial
   “Calcula o fatorial de n”
   [n]
   (if (= n 0)
      1
      (* n (fatorial (- n 1)))))

Problemas com essa implementação ?

(fatorial 5000)
java.lang.StackOverflowError
```

Funções Recursivas - loop/recur for the rescue

```
user> (doc loop)
-------------------------
clojure.core/loop
  (loop [bindings*] exprs*)
Special Form
  Evaluates the exprs in a lexical context in which the symbols in
  the binding-forms are bound to their respective init-exprs or parts
  therein. Acts as a recur target.

  Please see http://clojure.org/special_forms#loop
```

```
user> (doc recur)
-------------------------
recur
  (recur exprs*)
Special Form
  Evaluates the exprs in order, then, in parallel, rebinds
  the bindings of the recursion point to the values of the exprs.
  Execution then jumps back to the recursion point, a loop or fn method.

  Please see http://clojure.org/special_forms#recur
```

Fatorial com TCO

```
(defn fatorial-tco
   “Calcula o fatorial de n com TCO”
   [n]
   (loop [novo  n
          accum 1]
      (if (zero? novo)
         accum
         (recur (- n 1) (* accum novo)))))

(defn fatorial2-tco
   “Calcula o fatorial de n com TCO - interface com 2 args”
   [n
    accum]
   (if (zero? n)
      accum
      (recur (- n 1) (* accum n)))))
```

Exercício

- Implemente a função pow que recebe dois parâmetros, x e n, e retorna x^n
- Faça uma versão sem TCO e outra com TCO

1. Condição básica: $a^0 = 1$;
2. Relação de recorrência: $a^n = a \cdot a^{(n-1)}$, para $n > 1$ ou $n = 1$.

Desafio 1

Implementar o jogo adivinhe-o-número. Um número entre 1 e 100 é sorteado pelo computador e o jogador terá X chances de acertar esse número. A cada chute o jogo deverá diminuir o intervalo de chutes. Se as vidas acabarem, o jogo termina.
Ex.:
#segredo = 76

```
jogo: Digite um número entre 1 e 100:
usuário: 50
jogo: Digite um número entre 50 e 100:
usuário: 87
jogo: Digite um número entre 50 e 87:
usuário: 60
jogo: Digite um número entre 60 e 87:
usuário: 76
ACERTOU!!
```

```
(read-line)
para ler do teclado


(Integer/parseInt “10”)
converte string numérica em
inteiro
```

Estruturas de dados 101

- Clojure possui um conjunto rico de estruturas de dados
- **Listas, Vetores, Conjuntos e Maps**
- Clojure é uma linguagem *data oriented*, i.e., utiliza-se essas estruturas para representar entidades
- Todas as estruturas são imutáveis, ou seja, não podem ser destruídas, alteradas, etc…
- **Alguém tem ideia de como isso funciona?**
- https://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey/

Ideal Hash Trees - Phil Bagwell

Ideal Hash Trees

Phil Bagwell

Hash Trees with nearly ideal characteristics are described. These Hash Trees require no initial root hash table yet are faster and use significantly less space than chained or double hash trees. Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches. Array Mapped Tries(AMT), first described in Fast and Space Efficient Trie Searches, Bagwell [2000], form the underlying data structure. The concept is then applied to external disk or distributed storage to obtain an algorithm that achieves single access searches, close to single access inserts and greater than 80 percent disk block load factors. Comparisons are made with Linear Hashing, Litwin, Neimat, and Schneider [1993] and B-Trees, R.Bayer and E.M.McCreight [1972]. In addition two further applications of AMTs are briefly described, namely, Class/Selector dispatch tables and IP Routing tables. Each of the algorithms has a performance and space usage that is comparable to contemporary implementations but simpler.

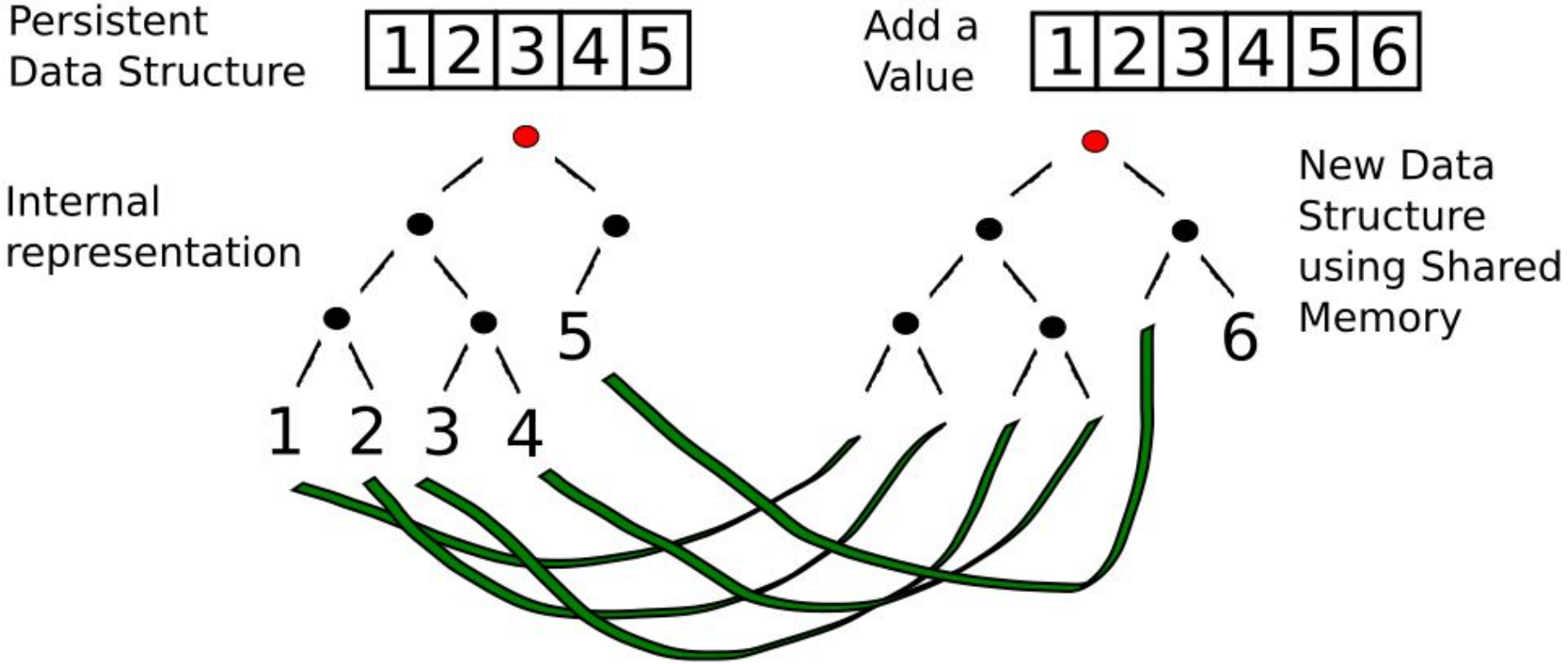
Categories and Subject Descriptors: H.4.m [**Information Systems**]: Miscellaneous

General Terms: Hashing,Hash Tables,Row Displacement,Searching,Database,Routing,Routers

---

Purely Functional Data Structures

Chris Okasaki


Persistent
Data Structure
1 2 3 4 5
Add a
Value
1 2 3 4 5 6
Internal
representation
New Data
Structure
using Shared
Memory
1 2 3 4
5
6

Estruturas de dados 101 - Vetores

- Vetores em Clojure são representados por colchetes []
- **[1 2 3]** é um vetor
- Repare que os elementos não precisam ser separados por vírgulas
- Eles são otimizados para inserção no final
- API Básica de vetores:
  - **vector**: cria um vetor (vector 1 2 3)
  - **count**: retorna a quantidade de elementos
  - **first**: retorna o primeiro elemento ou nil
  - **rest**: retorna a cauda da lista (tail)
  - **empty?:** retorna true se o vetor for vazio, false caso contrário
  - **last:** último elemento do vetor
  - **butlast:** todos menos o último

Estruturas de dados 101 - Vetores

```
(def nums [10 20 30 40 50])
(first nums) => 10
(last nums) => [20 30 40 50]
(count nums) => 5
(empty? nums) => false
(last nums) => 50
(butlast nums) => [10 20 30 40]

(conj nums 60) => [10 20 30 40 50 60]
(conj [1 2] 3) => [1 2 3]
(conj [] 1) => [1]
(conj nil 1) => [1]
```

Desafios

1. Escreva uma função **pilindromo** que recebe um vetor e retorna true se o mesmo for palíndromo, false caso contrário
2. Escreva uma função chamada **contem?** que recebe umm vetor e um número. A função deve retornar true se o número estiver no vetor e false caso contrário.


LAMBDA
LAMBDA
LAMBDA
ΛΛ
Λ
LAMBDA
ADAMS COLLEGE 1984

(λambda Calculus em Clojure)

Existem basicamente duas maneiras de se criar uma função anônima em clojure:

1. (fn [param1 param2 ...]
   corpo-da-função)

2. #(corpo %1 %2...)

λambda Calculus

Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f.\ \lambda x.\ x$
1	$1\ f\ x = f\ x$	$1 = \lambda f.\ \lambda x.\ f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f.\ \lambda x.\ f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f.\ \lambda x.\ f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\ \lambda x.\ f^{\circ n}\ x$

The addition function $\mathrm{plus}(m, n) = m + n$ uses the identity $f^{\circ(m+n)}$ (

$$\mathrm{plus} \equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

Alonzo Church

Estruturas de dados 101 - Vetores

- `map:` aplica uma função em cada um dos elementos do vetor, retornando um novo vetor modificado
- `filter:` filtra elementos de um vetor através de uma função predicado
- `reduce:` acumula valores de um vetor aplicando uma função binária ao acumulador, para cada elemento da lista*

Vetores - map

```
clojure.core/map
([f] [f coll] [f c1 c2] [f c1 c2 c3] [f c1 c2 c3
& colls])
```

Returns a **lazy sequence** consisting of the result of *applying f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the colls is exhausted*. Any remaining items in other colls are ignored. Function f should accept number-of-colls arguments. Returns a transducer when no collection is provided.

Vetores - map

Implementação em JS imperativo:

```
function mapeiaVetor(fun, vetor) {
    res = [];
    for(i = 0 ; i < vetor.length ; i++)
        res.push(fun(vetor[i]));
    return res;
}
```

```
vetor.map((x) => x+1);
mapeiaVetor((x) => x+1, vetor)
```

Vetores - map

Implementação recursiva em JS com Tail Call:

```
function mapeiaVetor(fun, vetor, res) {
    if(vetor.length===0)
        return res;
    else {
        res.push(fun(vetor[0]));
        return mapeiaVetor(fun,
vetor.slice(1),res);
}

mapeiaVetor((x) => x+1, vetor)
```

Vetores - map

Vamos implementar nossa própria versão de map ?

```
(defn mymap
  [f coll]
  … )
```

Vetores - map

```
(def nums [10 20 30 40 50 60 70 80 90 100])
```

```
(mymap ??? nums)
       ^^^
```

expressão lambda
função de um argumento

Conhecendo a sintaxe do lambda, como você faria para adicionar 5 em cada elemento do vetor ?

```
(mymap ??? nums)
(mymap ??? nums)
```

Vetores - filter

clojure.core/filter
([pred] [pred coll])

Returns a **lazy sequence** of the items in coll for **which (pred item) returns true.** pred ***must be free of side-effects***.Returns a transducer when no collection is provided.

Predicado é uma função booleana (retorna true ou false)

Vetores - filter

```
(def nums (range))

(filter odd? (take 10 nums))
(filter even? (take 10 nums))
(filter #(odd? %) (take 10 nums))
(filter #(even? %) (take 10 nums))
(filter (fn [n] ()) (take 10 nums))
```

Vamos implementar nossa própria versão de filter

```
(defn myfilter [pred coll] … )
```

Vetores - reduce

clojure.core/reduce
([f coll] [f val coll])
  f should be a function of 2 arguments. If val is not supplied,returns the result of applying f to the first 2 items in coll, then applying f to that result and the 3rd item, etc. If coll contains no items, f must accept no arguments as well, and reduce returns the result of calling f with no arguments. If coll has only 1 item, it is returned and f is not called. If val is supplied, returns the result of applying f to val and the first item in coll, then applying f to that result and the 2nd item, etc. If coll contains no items, returns val and f is not called.

Maps

Maps - intro

- Estrutura de dados do tipo chave/valor
- Principal ED para representar entidades
- As chaves, na maioria das vezes, são keywords
- Mas podemos usar qualquer valor
- Maps também são funções (???)

```
(def personagem
  {:nome “Bilbo”
   :raça “Hobbit”
   :habilidade :stealth
   :força 30})
```

Maps - intro

```
(def personagem
  {:nome “Bilbo Bolseiro”
   :raça “Hobbit”
   :habilidade :stealth
   :força 30})
```

(personagem :nome) => “Bilbo Bolseiro”
(:nome personagem) => “Bilbo Bolseiro”
(keys personagem) => (:nome :raça :habilidade :força)
(vals personagem) => (“Bilbo Bolseiro” “Hobbit” :stealth 30)

API básica

```
(def bilbo
  {:nome “Bilbo Bolseiro”
   :raça “Hobbit”
   :habilidade :stealth
   :força 30
   :equip {}})
```

```
(keys bilbo)
(vals bilbo)
(bilbo :raça)
(assoc bilbo :destreza 50)
(update bilbo :destreza inc)
(update bilbo :equip conj {:name “sting”})
(update-in bilbo
           [:equip :name]
           clojure.string/capitalize)
```

Sets

Sets - intro

Sets são estruturas de dados que representam conjuntos (teoria dos conjuntos)

Conjuntos não permitem elementos repetidos e não possuem ordem interna.

Sets - declaração

Existem duas formas para se definir um set:

```
> #{1 2 3 4 5}  ;;; literal

> (set [1 2 3 4 5]) ;;; por função
```

- Operações comuns em conjuntos:
  - união (set/union set1 set2)
  - intersecção (set/intersect set1 set2)
  - diferença (set/difference set1 set2)
- namespace clojure.set