

Linguagens de Programação Não-Convenicionais

Orlando

101

21/09/2015 ↴

Calendário: (dias com aula)

40% P1 ; 60% P2
85% P; 15% T 85% P; 15% T

Set: 21, 28

Out: 5, 19, 26

→ Prova P1

Nov: 9, ~~23~~, 30

dia 23 cancelou (por causa da SECCOMP)

Dec: 7, 14, 21

Jan: 4, 11, 18, ~~25~~ → Prova P2

Fériados, Recursos, etc: 12/10; 2/11; 16/11; 28/12

Lisp e Prolog

Permitem a introdução de programação para I. A.

Não significa
que Lisp e
Prolog moveram

Java, etc.
Agentes
DB..

O que é Lisp? Lisp é um "estilo", uma "família".

L Scheme (1974: Guy Steele Jr.)

L Common Lisp (1988: Guy Steele Jr.)

* L Clojure (2007: Rich Hickey)

Uma das contribuições do LISP é a sintaxe mínima.

↳ macros

✓ Clojure tem influências

↳ Lisp

↳ Java (polimorfismo/interfaces), (Interoperabilidade)

↳ ML/Haskell (imutabilidade)

↳ Script (agilidade)

Toda a máquina virtual Java está disponível em Clojure.

LISP é programação funcional

PROLOG é programação lógico/relacional

Ferramentas

- Oracle JDK

- LEININGEN

- GitHub p/ windows

- IntelliJ Idea 14 || Eclipse
+ Plugin Cursive || + Plugin Counter Clockwise || Light Table

Livros

- Programming Clojure - Chas Emerick Et. Al.

- Joy of Clojure 2.ed - Michael Fogus

Novo Projeto: CMD: lein new hello

Arquivo principal: project.clj

CMD: lein repl

CMD: CTRL+D (termina o repl)

LISP

102

↳ Também chamada de computação simbólica

- Símbolo (algo que tem um valor $a \rightarrow 3$)

- Expressão Simbólica (+ a 4)

Os parênteses definem sintaticamente uma expressão simbólica.

- função
- macros
- special forms



Operador
(símbolo)
RADOR

O +
Operando
RANDOS

(+ a 4) → programa, código

(1 2 3 4 5) → dado

A sintaxe, formato dos dados e programa é a mesma.
HOMOÍCÔNICA

Ex: (* (+ 3 4) (- 10 8))

$\underbrace{(+ 3 4)}_{\downarrow 7} \quad \underbrace{(- 10 8)}_{\downarrow 2}$

↓ 14

(def b 13) : cria um símbolo, b, que vai valer 13.

↳ quote (ou aportage) na frente do dado (1 2 3 4 5) para funcionar.

Ex: (reverse '(1 2 3 4 5))

Criar função

(defn toCelsius [t] ENTER

(, (* (- t 32) 5) 9))

↳ lembrar de fechar o parênteses do defn

C ... JAVA

LISP

toCelsius (7)

(toCelsius 7)

(defn max2 [a b] (if (> a b) a b))

(def max3 (fn [a b] (if (> a b) a b)))

lein Koman run

:a é uma keyword, um símbolo cujo valor é ele mesmo

28/09/2015 7

Programming Clojure, 2nd. (Aline na capa)

- Stuart Halloway

* lein new app lab

IntelliJ

Import!!! → com opçãoleinigen

→ para todo novo projeto grande!

Com o lein no CMD cria o projeto

Com a IDE (intelliJ) se edita

Abrir SRC → LAB → CORE.CLOJ

Run → Edit Configurations → + → Clojure Repl → Local

└ nomear como REPL

└ remover a sincronização e o make

- Escrever os programas / funções no arquivo

- Carregar elas no REPL

- Testar

-+ em C é binário

-? : → operador ternário em C

$(x > 3) ? 8 : 10$

if ↗ ↗
 yes no

→ quaternário

A Aridade é a quantidade de operandos em Clojure. (parcido com o conceito de overloads)

A grande vantagem do Clojure sobre os outros LISP é que ele "abraça" o JAVA.

Java
s.toUpperCase()
↳ string

Clojure
(.toUpperCase s)
↳ string

{ str não funciona com lista. Precisa usar o APPLY antes
 X (str "oi" "oi")
 ✓ (apply str "oi" "oi")
 ↳ pega o símbolo da função e aplica nos elementos da lista

A barra antes da letra denota que é um caractér

Motores da programação funcional: filter, map, reduce.

Notação de maps { } chaves (dicionários)

(def inventors { "lisp" "mcarthy" })

(get lab.core/inventors "lisp")

Um Keyword é um símbolo cujo valor é ele mesmo. Não precisa def.

Em Lisp guarda-se as palavras de variáveis grandes com Rifer.

Adiar P1 uma semana. Por causa da SECCOMP.

Adicionou novos campos aos mapas
(arvore item-menu :preco 0.71)

Lisp Clojure é uma linguagem dinamicamente tipada

Mudar de namespace

* ns* verifica o NS atual

↳ significa global (veio do common lisp)

(in-ns 'user) muda o namespace

require (carrega a biblioteca para se poder usar com o nome completo)
refer (pode usar a biblioteca com o nome reduzido).

Diferença de lista e vetor
(encadeada) ↳ tamanho certo
() []

O conj é polimórfico. Se a coleção é uma lista coloca no começo, se for um vetor vai no final.

{ first
rest
conj }

FAZER EXERCÍCIOS KOANS!!!

05/10/2015

04

Exercícios:

- Gere as listas:

- a) (1 10 2 30 3 50)
- b) (11 :a 12 :a 13 :a)
- c) (11 :b 12 :b 13)
- d) ((1 10) (2 30) (3 50))

funções:

- ↳ range
- ↳ repeat
- ↳ interleave
- ↳ interpose
- ↳ partition

- a) $\left. \begin{array}{l} (\text{def } a (\text{range } 1\ 4\ 1)) \\ (\text{def } b (\text{range } 10\ 60\ 20)) \end{array} \right\} (\text{interleave } (\text{range } 1\ 4\ 1) (\text{range } 10\ 60\ 20))$
 $(\text{interleave } a\ b)$
- b) $\left. \begin{array}{l} (\text{def } a (\text{range } 11\ 14\ 1)) \\ (\text{def } b (\text{repeat } 3 :a)) \end{array} \right\} (\text{interleave } (\text{range } 11\ 14\ 1) (\text{repeat } 3 :a))$
 $(\text{interleave } a\ b)$
- c) (interpose :b (range 11 14 1))
- d) (partition 2 2 (interleave (range 1 4 1) (range 10 60 20)))

* A estratégia da programação funcional é ter várias pequenas funções e combiná-las (como em Legos).

Duas novas opções de cruzar listas ao invés do inline.

INLINE: (partition 2 2 (interleave (range 1 4) (range 10 51 20)))

LET: (let [l1 (range 1 4)
 l2 (range 10 51 20)
 mix (interleave l1 l2)]
 (partition 2 2 mix))

} as variáveis locais podem ser utilizadas na função

THREAD: (\rightarrow) (range 10 51 20) } funcionamento
 MACRO (interleave (range 1 4)) } como pilha!
 (partition 2)

joga como
primeiro parâmetro

\rightarrow

joga como
segundo parâmetro

{ def cria variável global
 let cria variável local

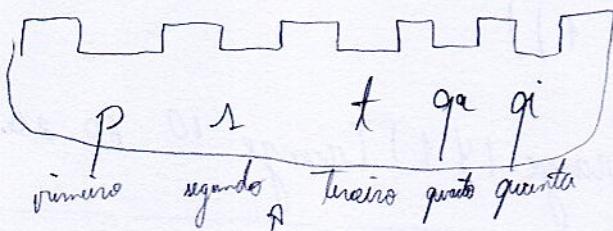
Destructuring

```
(def vals ["a" :name 43 "b" 43 0])
(def prim (first vals))
(def regur (second vals))
(def ter (nth vals 2))
```

Fazendo em uma linha só:

(let [[p r t q a q i] vals]

(str "os componentes não" p r t q a q i))



primeiro segundo terceiro quarto
 vai abrir o vetor e associar com
 o nome criado

Recebe uma estrutura (vetor) e
 abre ele em variáveis

I underline nos parâmetros de função significa don't care,
 para ignorar.

:as \rightarrow retém uma variável para o original.

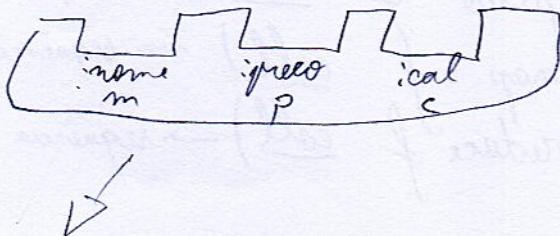
Destructuring para Maps

(def lanche { :nome "xburguer" :preco 9.99 :cal 600 })

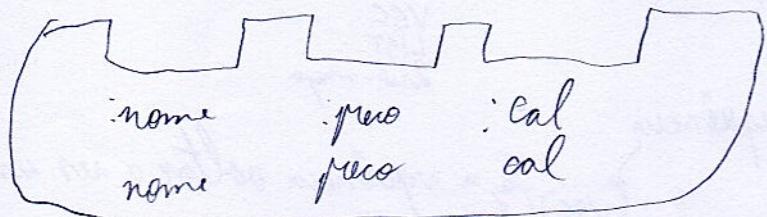
(: nome lanche)

(: preco lanche)

(: cal lanche)



(let [{ n : nome p : preco c : cal } lanche]
(str "os campos são: " n p c))



(let [{ :Keys [nome preco cal] } lanche]
(str "os campos são: " nome preco cal))

* { Filter
Map
Reduce }

MAP: → aplica a função em todos os termos → a resposta

parece uma lista mas não é. É uma SEQUÊNCIA!!

(def palavras ["sandode" "libelula" "cristal" "cerveja" "lua" "boi"])

(map count palavras)

(map #(toUpperCase %) palavras)

Sequências em Clojure

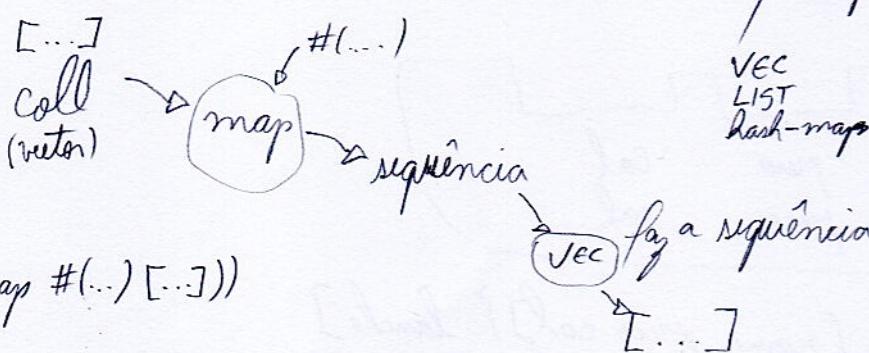
↳ Lazy

$\left\{ \begin{array}{l} (\text{filter } f \text{ } \underline{\text{coll}}) \rightarrow \text{sequência} \\ (\text{map } f \text{ } \underline{\text{coll}}) \rightarrow \text{sequência} \\ (\text{reduce } f \text{ } \underline{\text{coll}}) \rightarrow \text{sequência} \end{array} \right.$

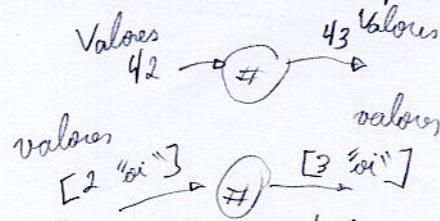
funções de ordem superior (pois aceitam funções como parâmetro)

- todas as coleções são sequências

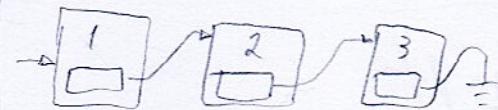
↳ é uma interface que todos seguem: listas, vetores, maps, etc.



Todas as sequências são imutáveis (persistentes) e Lazy



não valores diferentes! Eles não mudam de estado! Os dois existem! Ao modificar cria-se outro! É persistente.



Permite trabalhar com listas/insequências infinitas (não mexe com o dado quando ele é requisitado). Cura a medida que precisa.

O REPL "consome o lazy" por causa do print.

(def n2 (map #(do (println %) (inc %)) [1 2 3]))

→ não executa a função após o def.

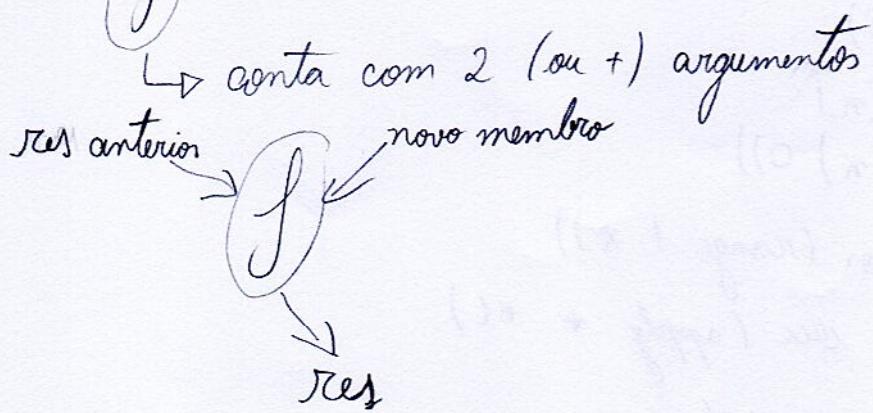
→ se colocar (take 1 n2) ai ele calcula

(cycle [2 4 8]) → trava, é infinito

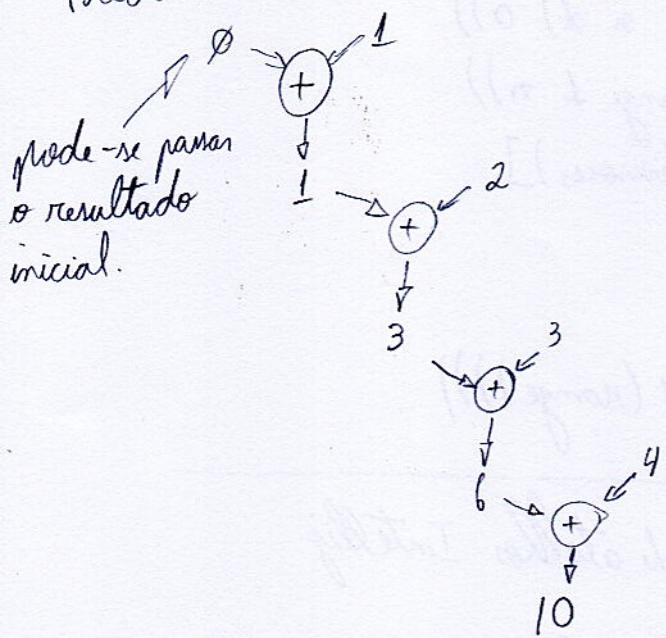
(def a (cycle [2 4 8])) → não trava, é lazy → não imprime

106
Reduce: (= fold = foldl ≠ foldr)

(reduce f) coll

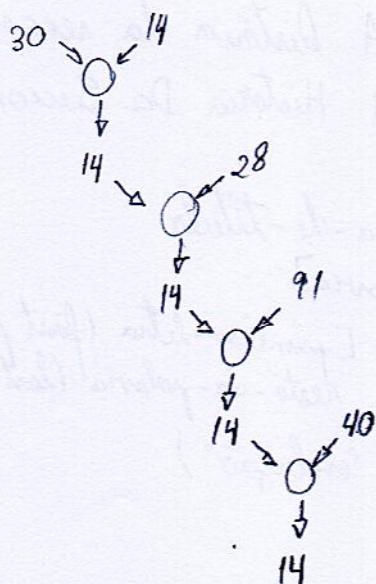
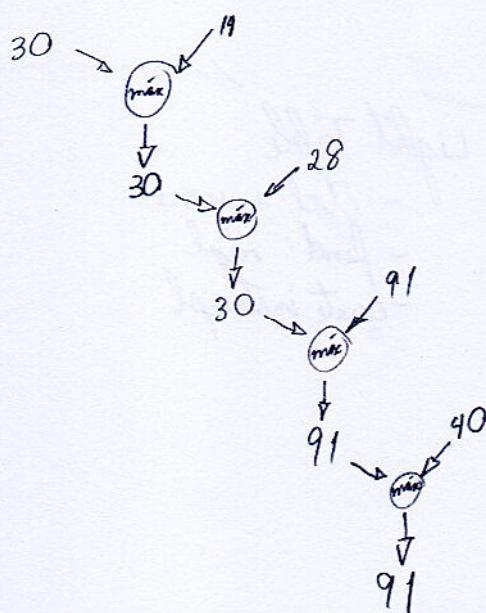


(reduce + [1 2 3 4]) → faz por iterações, um de cada vez



O map aplica a função nos elementos isoladamente. Já o reduce trabalha analisando os elementos.

(reduce max [30 14 28 91 40]) | (reduce min [30 14 28 91 40])



Problema: encontrar números perfeitos (a soma dos divisores é igual à ele mesmo) (inclui 0 e exclui de mesmo).

```
(def x 28)
(range 1 28)
(defn divisor [n]
(= (rem x n) 0))
(filter divisor (range 1 x))
(reduce *1) von (apply + *1)
```

```
(defn numero-perfeito [n]
(let [divisor (fn [d] (= (rem n d) 0))
      divisores (filter divisor (range 1 n))
      soma-divisores (apply + divisores)]
  (= soma-divisores n)))
```

```
(numero-perfeito 28)
(take 13 (filter numero-perfeito (drop 1 (range)))))
```

26/10/2015

* Configuração de atalhos IntelliJ

Exemplo: Título

maisvalos ↓ A história da seccomp
↓ A História Da Seccomp

{
(defn palavra-de-título
[palavra]
(let [primeira-letra (first palavra)
 resto-da-palavra (rest palavra)]))
→ (first "evolução") → o resultado é a letra "e"

Light Table
- ctrl + space
- find: repl
- create instance

Palavras mais usadas.

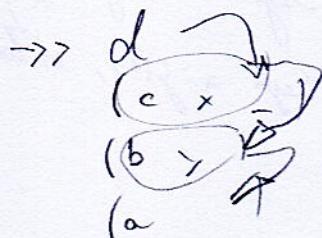
(digital)

- Fazer a lógica como se fosse no REPL
- Plano Top-Down
 - ↳ Ver os fluxos

Plano:

- Deixar tudo minúsculo
- Remover pontuação
- Separar as palavras
- Ordenar palavras
- Contar [-palavra, -quantas vezes]
- Ordenar quantidade
- Pegar n elementos
- Formatar
- Juntar

Thread Macro



Pega o resultado da linha 1
joga na próxima

↳ imprimir se for igual.

Trazer o anagrama na próxima aula!

07

*

Trabalho 1: Ordenar MP3

II 2: Mail Box

Anagramas:

- Carregar Arquivo Linha a Linha
- Receber palavra entrada
- Ordenar palavra entrada em ordem crescente caractere a caractere
- ~~Processar~~ Filtrar arquivo por palavras de tamanho igual
- Para cada palavra:
 - ↳ Salvar cópia Temp
 - ↳ Ordenar crescente
 - ↳ Verificar igualdade com a entrada

Exemplo: simular uma planilha

| Descrição | Estoque | Vendidos | Devolvidos |
|-----------|---------|----------|------------|
| Caneta | 500 | 150 | 10 |
| Borracha | 1200 | 370 | 3 |
| Total | 1700 | 520 | 13 |

Feito no PC

Let:

(defn *várias* [v])

```
(let [prim (:first v)
      seg (:second v)
      p1 (:família prim)
      p2 (:família seg)]
    (= p1 p2))
```

.....

MAP FILTER REDUCE

pegar uma função e aplicar em
toda a sequência

pega o predicado e deixa passar
alguns e remove outros.

pega dois elementos de cada
vez e faz uma conta (na
próxima iteração usa o resul-
tado anterior).

09/11/2015 ↴

Dia 14 de dezembro não haverá aula (motivo: vestibular)

Enunciado
Etapas: Fluxo
↓
REPL
↓
Função!

Let: "Seja $a = g(x), f(x)$ "
"Seja $a = g(x)$, então $f(x) \dots$ "

Melhor opção!

Pra testar no REPL.

Tools → REPL → Load File on
REPL

(Let [*variáveis locais*] (execução))
↳ uma variável para cada linha do REPL

PL dia 07/12!!!

30/11/2015

108

Dia 14 é feriado (vestibular) - não haverá aula.

Dados imutáveis é uma das características mais importantes e poderosas de cláusula.

As estruturas de dados do cláusula, por debaixo dos panos, são árvores.

atom = caixa

↳ a caixa é imutável, mas o que tem dentro da caixa pode ser mudado.

O bloco "do" tem como valor a última função

► coisas que precisam ser atualizadas são utilizados atoms.

18 da lista: arquivo de nome "Item 18 da lista impressa.txt" no PC.

Exercício: 3 tabelas

| Alunos | Nome |
|--------|-----------|
| 11 | Weslleyne |
| 13 | Adolfo |
| 15 | Alemanio |

Avaliações

| Código | Descrição | Peso | Data |
|--------|-----------|------|-------|
| P1 | "Prova" | 4 | 01/09 |
| T1 | Trabalho | 3.5 | 08/10 |
| P2 | Final | 2.5 | 25/12 |

Notas

| R.A | Código | Nota |
|-----|--------|------|
| 11 | P1 | 7.2 |
| 13 | P1 | 9.4 |
| 15 | P1 | 3.9 |
| : | : | : |

Criar uma função onde passa o R.A. → gera um relatório com a saída

→ 15

→ { : nome "Weslleyne"
: notas { "Prova" 7.2
"Trabalho" 3.9
"Final" 8.0 }
: media 5.3 }

A tabela será um vetor de map

Digitalizar e mandar por e-mail:

21/12/2015 ↴ (Após prova)

- Recursão em Listas!

(defn function [coll] → (if (empty? coll)
 (first coll)
 "parada"
 (function (rest coll))
 "éste é o geral")
)

Como parar a recursão:

- quando atinge a lista vazia

04/01/2016 ↴

(defn fat-iter [n fat]
 (if (zero? n)
 fat
 (fat-iter (dec n) (* n fat))))

(defn fat [n]
 (fat-iter n 1)) FACHADA

11/01/2016 ↴

Prolog → Normalmente voltada para bancos de dados.

Idéia de verdade / lógica; True / False

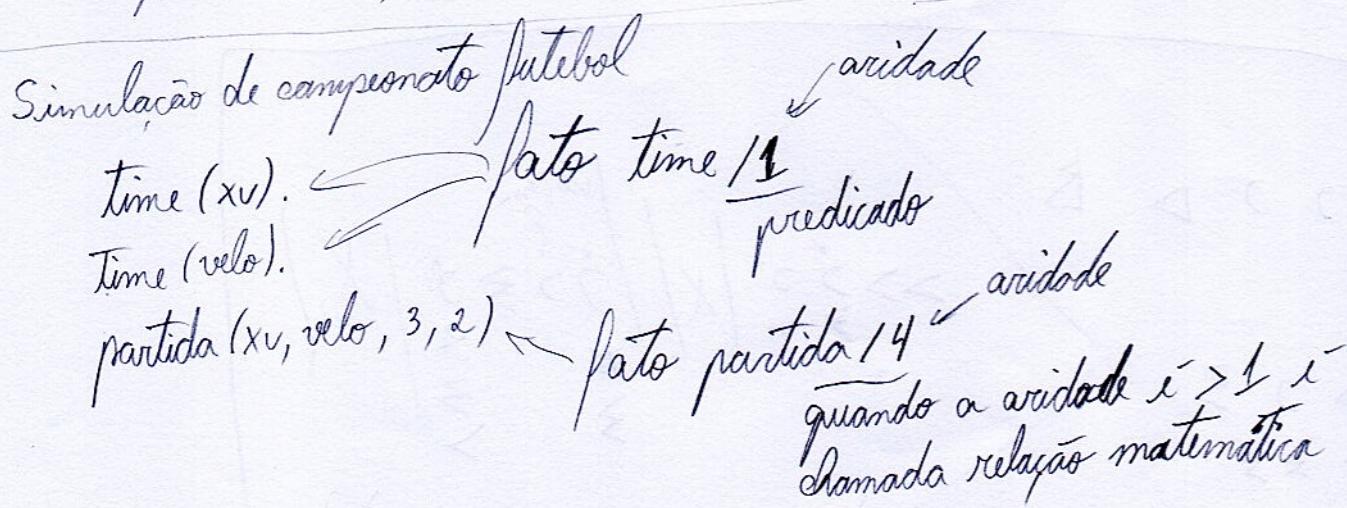
- Executar Prolog (interpretador Prolog)

- METAS: provar coisas

↳ existem queries

[09]
- variável é qualquer coisa que começo com maiúsculo!
compra (pat, X).

- Entrar com ";" mostra outras respostas da query
compra (X, sabao).
- a vírgula ";" é um "é" lógico
compra (X, Y), alimento (Y)



Interpretador

? - time (xv). ↗ meta
true ↗ consulta (query)
? - partida (xv, velo, X, Y)
X = 3
Y = 2

Regras:

compra - alimento :- $\overline{\overline{SE}}$ compra (X, Y), alimento (Y).

vence (A, B) :- partida (A, B, X, Y), X > Y.

Tem que ter ponto ":" no final da linha sempre

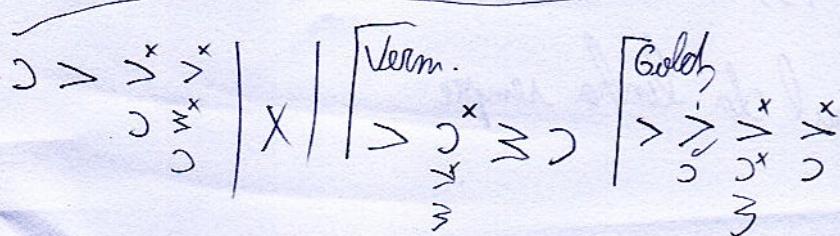
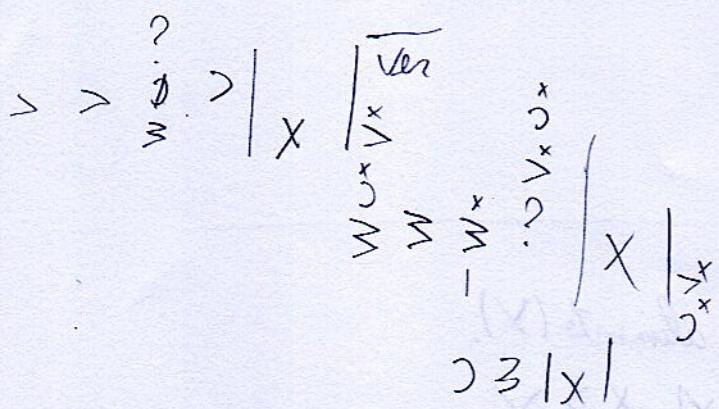
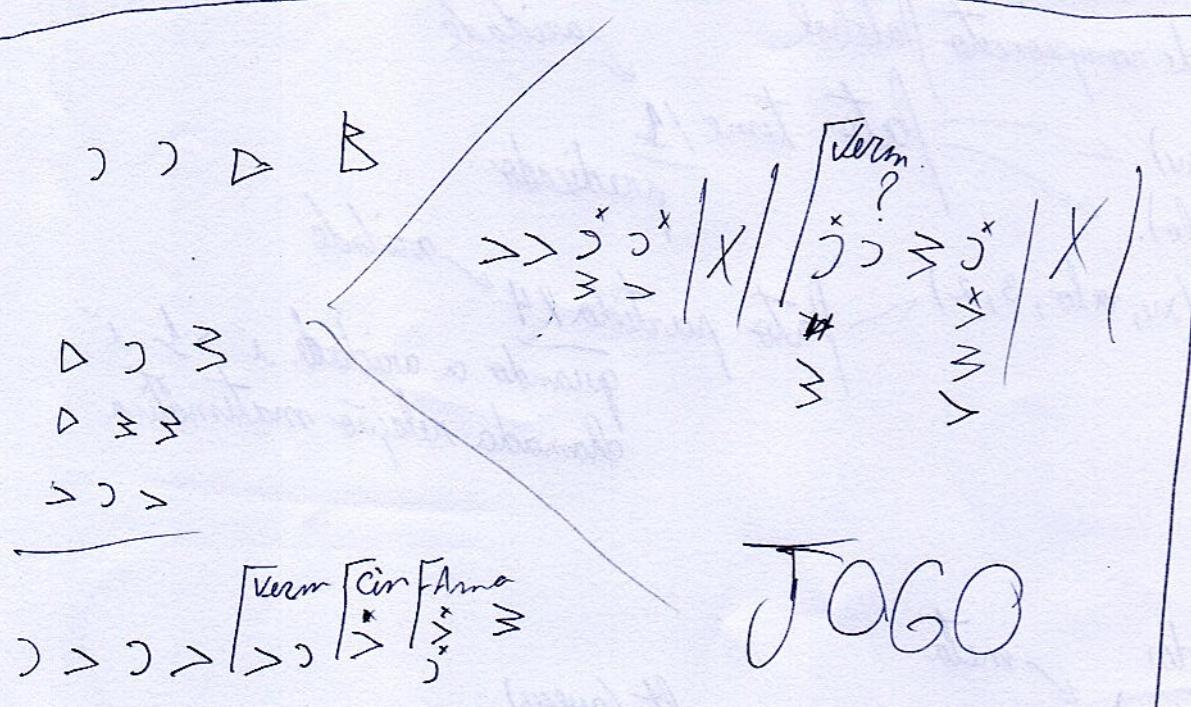
POJ... O que diferencia o Prolog de outras linguagens é a "unificação"

Programação relacional!

BACKTRACKING

↳ Classe de algoritmos

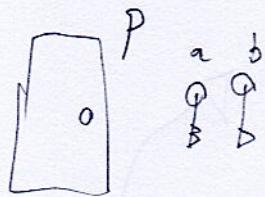
↳ Está embutido dentro da linguagem



backtracking · pl

[10]

$p(a)$
 $p(b)$
 $q(c)$
 $q(d)$
 $r(a, d)$



? $p(X), q(Y).$
? $p(X), q(Y), r(X, Y).$

} Este é o corredor!

↳ Quando trava volta para a escolha passada mais recente e muda, ai tenta novamente. Isto é o backtracking (volta na tentativa e erro e tenta novamente).

$_ (w) :- r(z, w), q(w).$

? $_ (A).$

Underscore (é uma variável, mas "não estou nem aí pra ela").

$_ (w) :- r(_, w), q(w).$

? $_ (A)$

$_ (W) :- r(_, W), !, q(W)$

↳ é o cut, não deixa voltar (é o inimigo no jogo).
na programação real o corte é algo bom. Pois já exclui as chaves que interessa, naquele ponto já se encontrou algo suficiente. Se depender de outra coisa é dall pra frente.

- No jogo a alavanca é o OV!

$_ (W) :- p(W)$

↳ quando existe duas regras.

$t(M) :- p(M), !, \text{fail}.$

not $p^{!!!}$

$t(M).$

Se não entra neste caso então OR , só é verdade! Esta é a operação NOT

O corte quebra o OR, se entra não volta mais!

18/01/2016 ↴

Prolog

- unificação
- relação
- query

royal family prolog

pai(joao, pedro).

pai(pedroI, pedroII).

pai(pedroII, isabel).

* Nova regra "avô" *

avô(x, y) :- pai(x, w), pai(w, y).

* Nova regra "irmao" *

irmao(x, y) :- pai(w, x), pai(w, y)

* "tio" *

tio(x, y) :- pai(w, y), irmao(w, x).

usuário (rui)

usuário (rei)

livro(o_almimista, machado-de-assis-edusp-2016)

empréstimo(o_almimista, rui, 13022016)

?- query

ver base de dados reflexiva

Reflexão em Listas

Tudo em Prolog é uma afirmação, uma sentença.

11

* Listas em Prolog:

$[] \rightarrow \text{vazia}$
 $[H | T] \rightarrow \text{cons}$
↓
1º elemento $[A, B, C | R]$
↓ resto 1º 2º 3º restos

| Parâmetros em LISP | Retorno em LISP | Prolog |
|--------------------|-----------------|---------------------|
| a_1, \dots, a_n | booleano | a_1, \dots, a_n |
| a_1, \dots, a_n | R | $a_1, \dots, a_n R$ |

?- append([1, 2], [30, 40], X).
?- append(X, Y, [1, 2, 3, 4, 5, 6]).

Exercícios: Sum

↳ Se o resultado não é booleano, além da lista é preciso o parâmetro de saída (a soma em si).

Stammer (repete)

Alternate

List_Ref (da a lista e o index).

Problemas com o NOT

Problema Prolog
Hipótese do mundo fechado.
↳ Só é verdade o que consta na base de dados.

Duas regras é um OUV! (a não ser que tente um CUT)

A ordem das regras muda tudo.
A ordem da base de dados também!

Recursão em Prolog
1. 5 ponto na prova.

Foco P2: Recursão
Listas em LISP e iterativo (5 pontos)

Prolog (5 pontos)

↳ base de dados

↳ regras novas (relações como avô e tio tendo pai apenas).

↳ querseys

↳ Backtracking (jogo - questão vai dar sentença e pedir resposta: exemplos de cut)

* Conjuntos não repetem elementos

programação funcional desde o básico

1) Relacione as colunas. (number? [1 2 3 \a :z 10 20 \b]) => col 2
col 1

filter
remove
take-while
drop-while
some

(\a :z 10 20 \b)
true
(1 2 3 10 20)
(\a :z \b)
(\a :z 10 20 \b) (1 2 3)

2) Relacione as colunas. (filter col 1 '(1 \a -5 12 :b [:c :d] 0 -3)) => col 2

number? 1
char? 2
keyword? 3
coll? 4

3 (:b)
4 (:c :d)
1 (1 -5 12 0 -3)
2 (\a)

3) Relacione as colunas. (filter col 1 '(1 -2 7 10 -5 0 29 41)) => col 2

zero? 1
odd? 2
even? 3
pos? 4
neg? 5

2 (1 7 -5 29 41)
4 (1 7 10 29 41)
5 (-2 -5)
1 (0)
3 (-2 10 0)

4) Relacione as colunas. (remove col 1 '([1 2] \b {:m 0} () 7 d :z)) => col 2

#(and (number? %) (odd? %)) 1
#(or (symbol? %) (list? %)) 2
#(and (map? %) (contains? % :m)) 3
#(or (vector? %) (char? %)) 4

2 ([1 2] \b {:m 0} 7 :z)
4 ({:m 0} () 7 d :z)
1 ([1 2] \b {:m 0} () d :z)
3 ([1 2] \b () 7 d :z)

5) Relacione as colunas. col1 => col2

(conj [\x \y] \a) 1
(conj '(\x \y) \a) 2
(cons \a [\x \y]) 3

3 (\a \x \y)
2 (\a \x \y)
1 [\x \y \a]

6) Relacione as colunas.

(into col 1 '(0 2 4 6)) => res
col 1

^-- col 2 são os elementos de res na ordem

[1 2] 1 → feitos 1 por vez, por lista
'(1 2) 2 → vai as contribuições
#{1 2} 3 → vai as contribuições

3 0 1 4 6 2
1 1 2 0 2 4 6
2 6 4 2 0 1 2

7) Relacione as colunas. (col 1 vector [0 1 2]) => col 2

map 1
mapv 2 → aplica de novo
mapcat 3 → mapa e concatena
map-indexed 4 → indexa depois

4 ([0 0] [1 1] [2 2])
1 ([0] [1] [2])
2 [[0] [1] [2]]
3 (0 1 2)

8) Relacione as colunas. (take 5 col 1) => col 2

(repeatedly rand) 1
(repeat (rand)) 2
(iterate rand 10) 3

1 (0.87 0.93 0.42 0.40 0.53)
3 (10 3.98 1.62 0.52 0.51)
2 (0.75 0.75 0.75 0.75 0.75)

9) Qual a diferença?

(for [x [1 3 5 7 2 4 6 15 17 19]] x) → realiza sempre
(for [x [1 3 5 7 2 4 6 15 17 19] :when (odd? x)] x) → realiza quando é ímpar
(for [x [1 3 5 7 2 4 6 15 17 19] :while (odd? x)] x) → até o primeiro par, ai para

1 3 5 7 2 4 6 15 17 19
1 3 5 7 15 17 19
1 3 5 7

- aplica a todos → Sim, é a mesma coisa!
- 10) (reduce #(conj % (f %2)) [] c) corresponde a (map f c) ?
- 11) Qual o resultado? 8
→ o vetor vai pra resposta ser um vetor
→ [c1 c2] → [c1 c2 c3]
- (reduce (fn [acc v] (if (> v acc) acc (+ 3 v))) [10 9 8 5 4 5])
→ 13
→ 11
→ 8
→ 7
→ 8 //
- Explique.
- 12) Relacione as colunas. col1 => col2
- (map #(vector % %2) [1 2 3] [\a \b])
(for [x [1 2 3] y [\a \b] :when (even? x)] (vector x y))
(reduce #(conj % (vector 1 %2)) '() [\a \b])
→ para uma lista inicial pra resposta ser uma lista
→ ([1 \b] [1 \a]) 3
→ ([1 \a] [2 \b]) 1
→ ([2 \a] [2 \b]) 2
- 13) Defina filter com base em reduce.
- 14) apply : qual a diferença?
→ Nenhuma
(max 4 7 2 8 0 5 1)
(apply max [4 7 2 8 0 5 1]) → aplica a função na coleção
- 15) Relacione as colunas. col1 => col2
- (conj [\x \y] \a)
(conj '(\x \y) \a)
(cons \a [\x \y])
→ (\a \x \y) 2
→ (\a \x \y) 3
→ [\x \y \a] 1
- 16) Relacione as colunas. (_____ col 1 {:a 1 :b 2} _____ col 2) => {:a 2 :b 2}
→ 2 :a inc
→ 1 :a 2
→ 3 [:a] inc
- 17) Dado
{:x {:m 3, :n 6}, :y 4, :z {:i {:f -1, :g 0}, :k 2}} a
Como gerar
{:x {:m 3, :n 6}, :y 4, :z {:i {:f -1, :g 1}, :k 2}} (update-in a [:z :i :g] inc)
usando uma única chamada de update-in?
- 18) Dado um vetor de maps de coordenadas, filtrar aqueles que colidem. Há colisão quando $|x_2 - x_1| < 30$ e $|y_2 - y_1| < 30$.
- Exemplo
(remover-colisões [{:x 20 :y 80} {:x 75 :y 70} {:x 60 :y 50}])
→ [{:x 20 :y 80}]
→ (defn abs [a] (if (< a 0) (* a -1) a))
→ vetor
- Projeto do fluxo de dados
- Criar um predicado colide que compara dois maps num vetor.
 - (colide [{:x 100 :y 200} {:x 50 :y 200}]) cart: (for a V b V :when (not= a b))
 - ;;= false
 - (colide [{:x 10 :y 10} {:x 20 :y 15}])
 - ;;= true
 - Fazer o produto cartesiano dos maps, sem (a,a).
 - Obter colisões
 - Coletar maps das colisões.
 - Removê-las do vetor de maps original.
- (defn colide [a]
→ (if (< △ 30)
→ (abs (- (:x (first a)) (:x (second a))))
→ (if (< △ 30)
→ (abs (- (:y (first a)) (:y (second a))))
→ True
→ False) False))
- Sessão de REPL
é com você!
- unesp <> igce <> linguagens de programação não convencionais <> roteiros 2