

# Programação em Lógica

Programação Declarativa: o programador descreve o que quer que o resultado seja      Ex: SQL

Dado um problema, em vez de projetar e escrever um algoritmo para o resolver, simplesmente especifica-se o problema e o computador resolve-o

Vantagens:

1. prototipagem rápida
2. código pequeno
3. flexível e intuitivo

PROLOG: linguagem de programação em lógica (não 100% declarativa)

- Os programas não descrevem de conhecimento/relações sob a forma de predicados de lógica de primeira ordem
- A computação começa com uma "query" e o computador tenta prová-la



# PROLOG

Programa: conjunto finito de predicados

usam factos e regras para expressar conhecimento como relações

Computação: para de um objetivo generalizações de funções

Programa Correto: não permite a dedução de factos indesejados

Programa Completo: permite a dedução de tudo o que desejado

Tudo em PROLOG é um termo

Termo

- Contante: número OU átomo (minúscula)
- Variável: começam com maiúscula ou underscore (-)
- Composto: functor e argumentos  
none/aridade  $\Leftrightarrow$  átomo/número de argumentos

Facto: relação que é verdade male (hom.).

A interpretação/semântica tem que ser definida e partilhada : ...

Regra: permite a dedução de novo conhecimento a partir do existente

Cláusula de Horn: Cabeça :- Corpo

Conjunção: , (e)

Disjunção: ; (ou)

- As regras têm uma interpretação declarativa e procedural
- A cabeça de uma regra pode ter zero ou mais argumentos
- As variáveis não são instanciadas imediatamente (CUIDADO)

Y Lucy: ter duas respostas possíveis (sim/não)  
↓ pode incluir variáveis — quantificadas existencialmente — → anónima

Mundo Fechado: tudo o que não puder ser deduzido é mentira

Notas:

- Usar unificação implícita!
- Colocar argumentos de input antes de argumentos de output

= operador de unificação      | = "não unificável"

Regras: cláusula de Horn completa

(nó cabeça)

Facts: cláusula de Horn em que o corpo é sempre verdadeiro

Regras: cláusula de Horn nem cabeça (nó e corpo)

Predicado: conjunto de cláusulas para o mesmo functor  
↳ factos ou regras

- A documentação deve incluir uma declaração do modo para cada argumento:
- + o argumento é instanciado quando o predicado é chamado INPUT
- o argumento não é instanciado nas chamadas ao predicado OUTPUT
- ? o argumento pode ser instanciado ou não IN/OUT

VERSATILIDADE

U: r-union- $\Delta$ ( $X_i$ ) :-  $r(X_i)$ ;  $\Delta(X_i)$

$\Delta$ : r-inter- $\Delta$ ( $X_i$ ) :-  $r(X_i)$ ,  $\Delta(X_i)$

X: r-times- $\Delta$ ( $X_i, X_j$ ) :-  $r(X_i)$ ,  $\Delta(X_j)$

$\Delta$ : r-join- $\Delta$ ( $X_1, X_2, X_3$ ) :-

IT: r-1-3( $X_1, X_3$ ) :-  $r(X_1, X_2, X_3)$

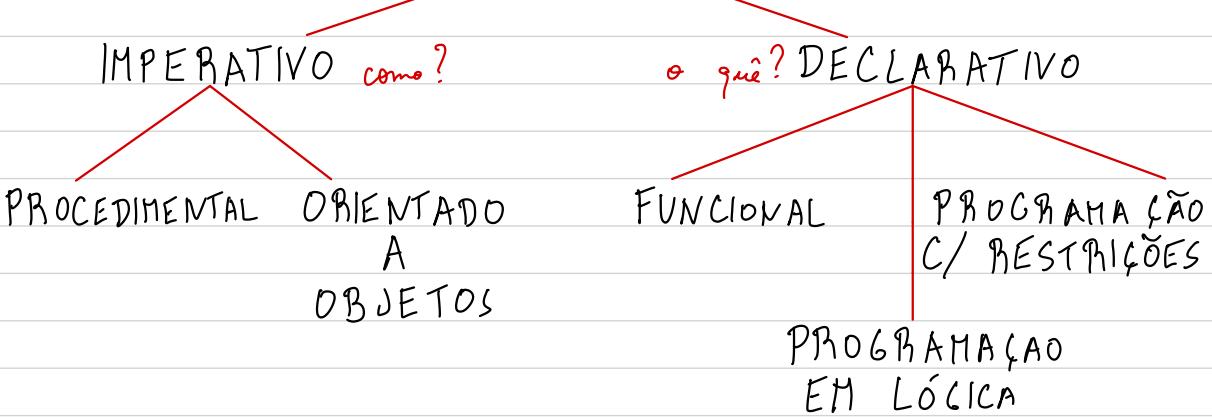
$\Delta$ : r-join- $\Delta$ ( $X_1, X_2, X_3$ )

R: r-1( $X_1, X_2, X_3$ ) :-  $r(X_1, X_2, X_3)$ ,  $X_2 > X_3$

$\Delta$ : r-minus- $\Delta$ ( $X_i$ ) :-  $r(X_i)$ ,  
 $\Delta(X_i)$

# TP 1

## PARADIGMAS



$@<$  para evitar soluções redundantes, impõe-se uma ordem no  $\otimes$ , através de uma comparação alfanumérica

• Para usar  $X > Y$ ,  $X$  e  $Y$  têm de estar instanciados

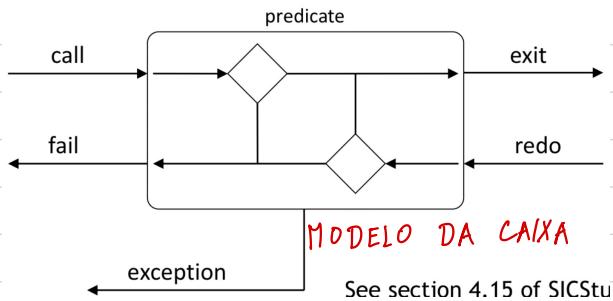
Exemplos Comuns:

1. first-alphabetical-child ( $P, C$ ) :- parent ( $P, C$ ),  $\backslash + \text{parent} (P, X)$ ,  $X @< C$ .
2. has-exactly-1child ( $P$ ) :- parent ( $P, C$ ),  $\backslash + ((\text{parent} (P, X), X) = C)$ .
3. has-exactly-2children ( $P$ ) :- parent ( $P, C1$ ), parent ( $P, C2$ ),  $C1 @< C2$ ,  $\backslash + ((\text{parent} (P, X), X) = C1, X = C2)$ .
4. has-2plus-children ( $P$ ) :- parent ( $P, C1$ ), parent ( $P, C2$ ),  $C1 @< C2$ ,  $\backslash + ((\text{parent} (P, C3), C3) = C1, C3 @< C2)$ .

NÃO USAR = & NÃO USAR ...  
MATILDE

# Como funciona Prolog?

1. de cima para baixo
2. da esquerda para a direita
3. com backtracking



See section 4.15 of SICStu

Trace: N S Your ID Profundidade Ponto : Objetivo ?

Aritmética: expressões aritméticas não são avaliadas imediatamente  
usam is (o lado direito tem que estar instanciado) ↗

$\ldots =:= \ldots$  avalia as expressões e se não iguais  $>, <$   
 $\ldots =\backslash= \ldots$  avalia as expressões e se não diferentes  $>=, = <$

$\ldots == \ldots$  verifica se os termos não são literalmente idênticos  $@<, @>$   
 $\ldots \backslash== \ldots$  verifica se os termos não são literalmente idênticos  $@>=, @<$

$X // Y$ : quociente inteiro truncado para 0

$X \text{ div } Y$ : quociente inteiro arredondado para baixo

$X \text{ rem } Y$ :  $X - Y * (X // Y)$

$X \text{ mod } Y$ :  $X - Y * (X \text{ div } Y)$

ORDEM MÁSICO

**Recursão:** baseada na prova por indução - lástimas base e recursivas

**"Tail Recursion":** adiciona um novo argumento (o acumulador) ao predicado, tornando a chamada recursiva a última chamada

### Recurridade Normal

$\text{numN}(0, 0).$

$\text{numN}(N, \text{Sum}) :- N > 0,$

$N1 \text{ is } N - 1$

$\text{numN}(N1, \text{Sum}1)$

$\text{Sum} \text{ is } \text{Sum}1 + N.$

### "Tail Recursion"

$\text{numN}(N, \text{Sum}) :- \text{numN}(N, \text{Sum}, 0).$

$\text{numN}(0, \text{Sum}, \text{Sum}).$

$\text{numN}(N, \text{Sum}, \text{Acc}) :- N > 0,$

$N1 \text{ is } N - 1,$

$\text{Acc}1 \text{ is } \text{Acc} + N,$

$\text{numN}(N1, \text{Sum}, \text{Acc}1).$

**Listas:**  $[..., \dots]$

**Lista Vazia:**  $[]$

• A representação interna usa o functor . e dois argumentos: a cabesa e a cauda da lista

• Strings não listas de códigos ASCII

**Lista:**  $[H | T]$   
  └── cabeca  
  └── cauda

• Depois do pai, vem sempre uma lista  
Ex:  $[4] = [4 | []]$

• Uma lista vazia é uma lista

• Um conteúdo de lista cuja cauda é uma lista é uma lista

# TP2

2. pairs ( $X, Y$ ) :-  $\downarrow(X)$ ,  $q(Y)$ .

pairs ( $X, X$ ) :-  $u(X)$ .

$u(1)$ .

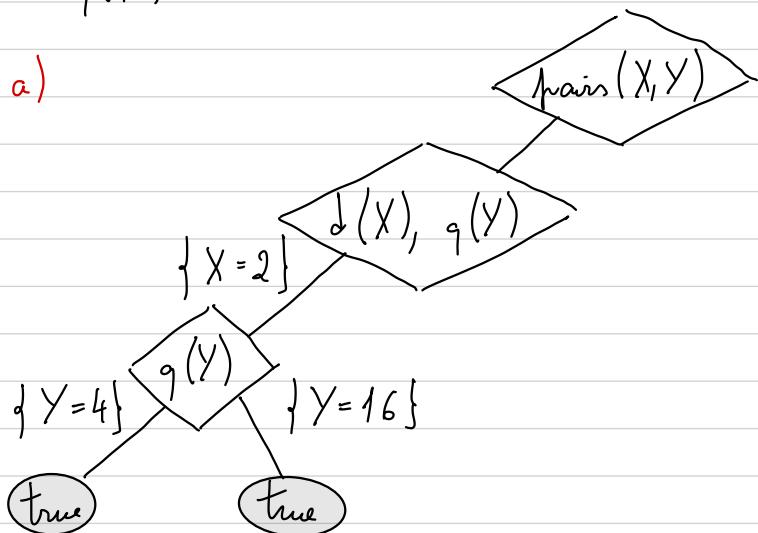
$\downarrow(2)$ .

$\downarrow(4)$ .

$q(4)$ .

$q(16)$ .

a)



length (?List, ?Lge): tamanho de uma lista flexível

→ length ([], 0).

length ([-|T], L) :-

length (T, L1),  
L is L1 + 1.

member (?Elem, ?List): numero da lista flexível

→ member (X, [X|\_]).

member (X, [-|T]) :- member (X, T).

memberchk (?Elem, ?List): verifica se é numero da lista determinista

→ memberchk (X, [X|\_]).

memberchk (X, [-|T]) :- X != Y, memberchk (X, T).

append (?L1, ?L2, ?L3): acrescenta duas numa terceira flexível

→ append ([], L2, L2).

append ((H|T), L2, [H|T3]) :- append (T, L2, T3).

sort (+List, -SortedList): ordena uma lista

keySort (+PairList, -SortedList): ordena uma lista de pares chave-valor

Biblioteca de Listas :- use-module(library(lists))

$\text{nth} 0 (? \text{Pos}, ? \text{List}, ? \text{Elem}) / \text{nth} 1 (? \text{Pos}, ? \text{List}, ? \text{Elem})$   
 $\text{nth} 0 (? \text{Pos}, ? \text{List}, ? \text{Elem}, ? \text{Rest}) / \text{nth} 1 (? \text{Pos}, ? \text{List}, ? \text{Elem}, ? \text{Rest})$   
 $\text{select} (? X, ? X \text{ List}, ? Y, ? Y \text{ List})$   
 $\text{delete} (+ \text{List}, + \text{To Del}, - \text{Rest}) / \text{delete} (+ \text{List}, + \text{To Del}, + \text{Count}, - \text{Rest})$   
 $\text{last} (? \text{Init}, ? \text{Last}, ? \text{List})$   
 $\text{segment} (? \text{List}, ? \text{Segment})$   
 $\text{sublist} (+ \text{List}, ? \text{Part}, ? \text{Before}, ? \text{Length}, ? \text{After})$   
 $\text{append} (+ \text{List Of Lists}, - \text{List})$   
 $\text{reverse} (? \text{List}, ? \text{Reversed})$   
 $\text{rotate-list} (+ \text{Start}, ? \text{List}, ? \text{Rotated})$   
 $\text{transpose} (? \text{Matrix}, ? \text{Transposed})$   
 $\text{remove-dups} (+ \text{List}, ? \text{Pruned List})$   
 $\text{permutation} (? \text{List}, ? \text{Permutation})$   
 $\text{sum-list} (+ \text{List Of Numbers}, ? \text{Sum})$   
 $\text{max-member} (? \text{Max}, + \text{List}) / \text{min-member} (? \text{Min}, + \text{List})$   
 $\text{max-member} (? \text{Compl}, ? \text{Max}, + \text{List}) / \text{min-member} (? \text{Compl}, ? \text{Min}, + \text{List})$   
  
 $\text{maplist} (: \text{Pred}, + L) / \text{maplist} (: \text{Pred}, + L_1, ? L_2) / \text{maplist} (: \text{Pred}, + L_1, ? L_2, ? L_3)$   
 $\text{map-product} (: \text{Pred}, + X_S, + Y_S, ? \text{List})$   
 $\text{scanlist} (: \text{Pred}, + X_S, ? \text{Start}, ? \text{Final})$   
 $\text{sumlist} (: \text{Pred}, + X_S, ? \text{Start}, ? \text{List})$   
 $\text{none} (: \text{Pred}, + \text{List}) / \text{none} (: \text{Pred}, + X_S, ? Y_S) / \text{none} (: \text{Pred}, + X_S, ? Y_S, ? Z_S)$   
 $\text{include} (: P, + X, ? L) / \text{include} (: P, + X, + Y, ? L) / \text{include} (: P, + X, + Y, + Z, ? L)$   
 $\text{exclude} (: P, + X, ? L) / \text{exclude} (: P, + X, + Y, ? L) / \text{exclude} (: P, + X, + Y, + Z, ? L)$   
 $\text{group} (: \text{Pred}, + \text{List}, ? \text{Front}, ? \text{Back})$

even (X):  $X \bmod 2 == 0$   
square (X, Y):  $Y \text{ is } X * X$

sum (A, B, C):  $C \text{ is } A + B$   
from (X, Y, Z):  $Z \text{ is } X ** Y$

# TP3

1.

a)  $[a \mid [b, c, d]]$

b)

c)

d)

e)

f)

g)

h)

i)  $[H, T] = [leic, T_{no}]$ .  
 $H = leic, T = [T_{no}]$

j)  $[Inst, func] = [gram, LEIC]$ .  
 $Inst = gram, LEIC = func$

k)  $[One, Two \mid Tail] = [1, 2, 3, 4]$ .  
 $One = 1, Two = 2, Tail = [3, 4]$

l)  $[One, Two \mid Tail] = [leic \mid Rest]$ .  
 $One = leic, Rest = [Two, Tail]$

## Funcionalidades Não - Lógicas

**Cut (!)**: é sempre bem - sucedido como um objetivo (pode ser ignorado numa leitura declarativa), limitando o Prolog a todas as escolhas / decisões tomadas desde que o objetivo - foi unificado com a cláusula onde está o cut

- Poda todas as cláusulas para o mesmo predicado abaixo daquela em que o cut está
  - Poda todas as soluções alternativas para os objetivos à esquerda do cut na cláusula
  - Não poda os objetivos já direta do cut na cláusula
- ↓
- Podem produzir várias soluções via backtracking
  - Backtracking para o cut falha e causa backtracking para o último ponto de escolha / decisão

**Cut Vermelho**: influencia os resultados — se se remove o cut, os resultados são diferentes

**Cut Verde**: não influencia os resultados, mas é usado para aumentar a eficiência — se se remove o cut, os resultados são os mesmos, mas o Prolog vai explorar ramos que não vão levar a nenhuma solução possível

Negação:  $\text{not}(X) :- X, !$ ,  $\text{fail}$ . → falha sempre  
 $\text{not}(\neg X)$ . → pregunta que segunda cláusula não é alcançada em backtracking

• Pode ser utilizado com valores instanciados

Condicional:  $\text{pred-ite}(If, Then, -Else) :- If, Then.$   
 $\text{pred-ite}(If, -Then, Else) :- \text{not}(If), Else.$

⇒

$\text{pred-ite}(If, Then, -Else) :- If, !, Then.$   
 $\text{pred-ite}(If, -Then, Else) :- Else.$

Input/Output: baseado em streams, usado para ler ou escrever, em modo texto (caracteres e termos) ou binário (bytes)

• Em cada momento, só há um stream atual de input e um stream atual de output — os predicados de I/O operam no stream atual correspondente

• Input e output não pode ser desfeito, mas o "binding" de variáveis (de predicados de input) é desfeito quando em backtracking

read/1 lê um termo

write/1 escreve um termo

see/1 abre ficheiro para leitura

tell/1 abre ficheiro para escrita

nl/0 imprime uma nova linha  
get/1 / put/1 / peek/char/cole/byte obtém / imprime

reen/0 fecha-o

told/0 fecha-o

use-module(library(...)); consult(..); ensure-loaded(..); include(..); [...]

repeat

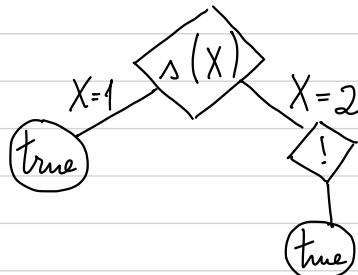
Between(+Lower, +Upper, ?N)

?Random

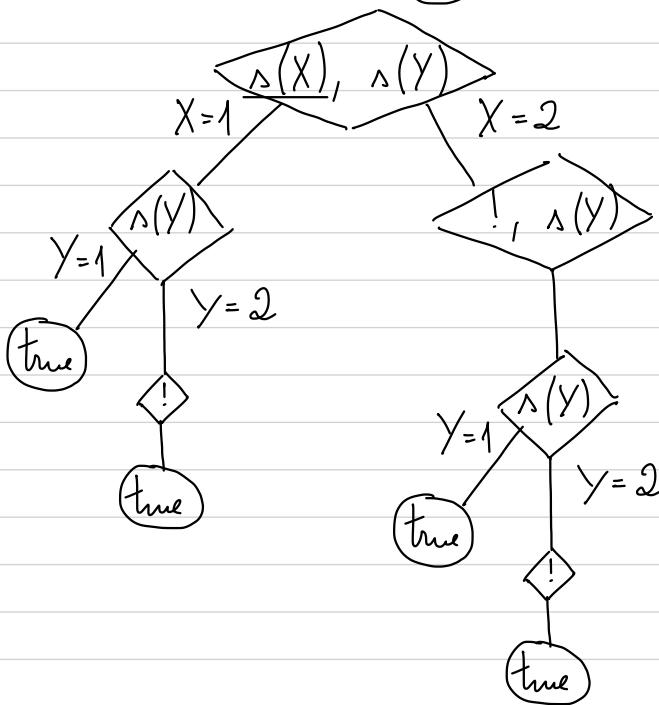
# TP 4

1.  $\Delta(1)$ .  
 $\Delta(2) :- !$ .  
 $\Delta(3)$ .

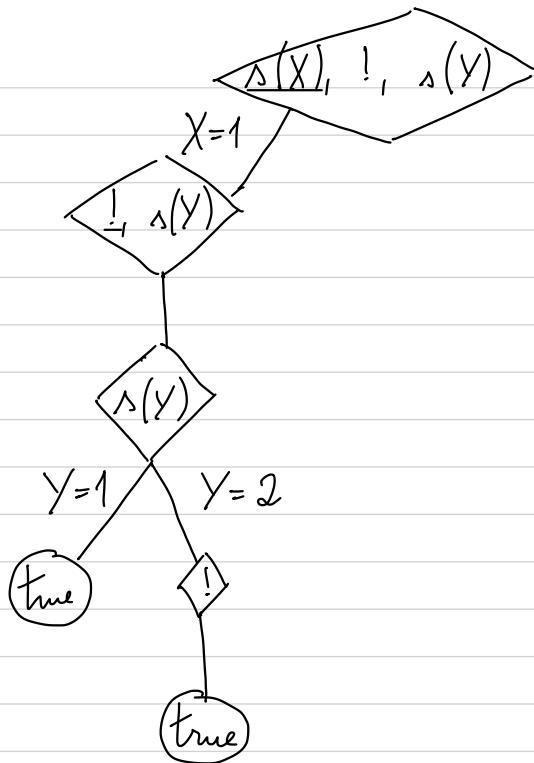
a)



b)



c)



3.

$\text{immature}(X) :- \text{adult}(X), !, \text{fail}.$   
 $\text{immature}(-X).$

$\text{adult}(X) :- \text{person}(X), !, \text{age}(X, N), N \geq 18.$   
 $\text{adult}(X) :- \text{turtle}(X), !, \text{age}(X, N), N \geq 50.$   
 $\text{adult}(X) :- \text{spider}(X), !, \text{age}(X, N), N \geq 1.$   
 $\text{adult}(X) :- \text{bat}(X), !, \text{age}(X, N), N \geq 5.$

# Unificação e Modelo de Execução

Terminos  $\left\{ \begin{array}{l} \text{"Ground": nem variáveis (completamente instanciados)} \\ \text{"Unground": com variáveis} \end{array} \right.$

Unificação: como o Prolog faz corresponder dois termos

Terminos Unificáveis: se são o mesmo, ou se o podem ser depois de substituição de variáveis

Substituição: conjunto de pares  $X_i = t_i$  onde

- (1)  $X_i$  é uma variável
- (2)  $t_i$  é um termo
- (3)  $X_i \neq X_j, \forall i \neq j$
- (4)  $X_i$  não ocorre em nenhum  $t_j, \forall i, \forall j$

• Aplica-se uma substituição  $\theta$  a um termo  $T$  ( $T\theta$ ) é substituir em  $T$  todas as ocorrências de  $X$  para  $t_i$ , para todos os pares  $X=t$  em  $\theta$

Instância: A é instância de B se existe uma substituição  $\theta$  tal que  $A=B\theta$

Instância Comum: um termo  $T$  é instância comum de  $T_1$  e  $T_2$  se existem substituições  $\theta_1$  e  $\theta_2$  tais que  $T=T_1\theta_1$  e  $T=T_2\theta_2$

Maior: um termo G é maior do que o termo T se T é uma instância de G mas G não é uma instância de T

**Variante:** um termo  $V$  é uma variante de um termo  $T$  se ele pode ser convertido noutro através de uma simples renomeação de variáveis.

Dadas duas frases atómicas,  $t_1$  e  $q_1$ , um algoritmo de unificação retorna uma substituição  $\theta$  (o unificador mais geral) que as torna idênticas (ou falha se essa substituição não existir)

$\theta$  é o unificador (mais geral) das duas frases

$$\text{Unify}(t_1, q_1) = \theta, \quad t_1\theta = q_1\theta$$

**Unificador Mais Geral:** aquele que comfomte as variáveis o mínimo possível — a respetiva instância é a mais geral

### Algoritmo de Unificação

inicializar  $\theta$  vazio

push  $T_1 = T_2$  na pilha

enquanto a pilha não estiver vazia:

top  $X = Y$  da pilha

caso:

$X$  é uma variável que não ocorre em  $Y$ :

substituir  $Y$  por  $X$  na pilha e em  $\theta$   
adicionar  $X = Y$  a  $\theta$

$Y$  é uma variável que não ocorre em  $X$ : ...

$X$  e  $Y$  são constantes ou variáveis idênticas: continue

$X$  é  $f(X_i)$  e  $Y$  é  $f(Y_i)$  para algum função  $f$ :

push  $X_i = Y_i$  na pilha

caso contrário: return falha

return  $\theta$

Státuos os termos não constantes? Unificam se não a mesma  
Um dos termos é variável? É instanciada para o outro termo  
Amplios os termos não variáveis? Ligan-se / limitam-se um ao outro

Termos compostos unificam se ...:

- (1) têm o mesmo functor e aridade
- (2) todos os argumentos correspondentes unificam
- (3) todas as substituições são compatíveis

"occurs check": verifica se  $X$  não ocorre em  $Y$  e vice-versa

PROLOG salta este passo FORÇAR unify-with-occurs-check/2

Programa: composto por cláusulas

Cláusulas: frases lógicas universalmente quantificadas

$A :- \_, B_1, \dots$

Computação: encontrar uma instância de uma dada query  $Q$  que é logicamente dedutível do programa  $P$

Query: conjunção existencialmente quantificada

$\dots, A_i, \dots$

Objetivo: átomo ou termo composto

• Dado um programa  $P$  e uma query inicial  $Q$ , a computação termina com sucesso se (uma instância de  $Q$  foi provada) ou nem sucesso se  $Q$  não pode ser provada

• A computação pode não terminar devido a regras recursivas

**Resolvente:** pergunta conjuntiva (query) com o conjunto de objetivos para aindaarem processados

**Trace:** evolução da computação (sequência de resolventes) com informação sobre o objetivo selecionado, a regra selecionada para redução e a substituição associada

**Redução:** substituição, na resolvente, do objetivo  $G$  com o corpo de uma cláusula cuja cabeça unifica com  $G$

### Algoritmo do Interpretador Abstrato

Seja  $\emptyset$  a resolvente

Enquanto a resolvente não estiver vazia:

1. Escolher um objetivo  $A$  para resolvente
2. Escolher uma cláusula renomeada  $B = \dots, B_i, \dots$  do programa tal que  $A \wedge B$  unificam com um MGU  $\theta$  (terminar se nenhum objetivo e cláusula existem)
3. Remover  $A$  da resolvente e adiciona  $\dots, B_i, \dots$
4. Aplica  $\theta$  à resolvente e a  $\emptyset$

Se a resolvente estiver vazia, retornar  $\emptyset$   
Senão, retornar falha

Uma implementação de Programação Lógica tem de instanciar o interpretador abstrato, tomando decisões (escolha do objetivo da resolvente, escolha da cláusula, adição do objetivo à resolvente) que influenciam como é que a computação é realizada

## PROLOG:

- (1) A escolha do objetivo da resoluente é da esquerda para a direita NÃO AFETA
- (2) A escolha da cláusula é de cima para baixo com backtracking AFETA
- (3) A adição de objetivo(s) à resoluente é no início ↗ DFS

- A resoluente pode ser vista como uma árvore com dados auxiliares (Pontos de backtracking)
- Uma árvore de pesquisa contém todos os caminhos de pesquisa possíveis

Play: query  $\emptyset$

Nós: resoluentes, com o objetivo selecionado

Frestas: cláusulas cuja cabeça unifica com o objetivo selecionado no nó de origem, incluindo substituições da unificação

Folhar: nós de nucoso ( $\times$  a resoluente vazia) ou de falha

Caminhos: computação de  $\emptyset$  pelo programa

• As árvores de pesquisa não são independentes do critério de seleção de cláusulas, mas dependem do critério de seleção de objetivos — o número de nós de nucoso é sempre o mesmo, mas a transição até eles depende da estratégia do interpretador

1. DFS: incompleta

2. BFS: completa

3. paralelismo OR: pesquisa todos os ramos da árvore em paralelo

4. paralelismo AND: executa todos os objetivos da resoluente em paralelo

• A árvore é importante para a eficiência!

# Colar Soluções

Como obter todas as soluções de uma query?

1. findall (+Template, +Goal, -List): encontra todas as soluções, incluindo repetições
2. bagof (+Template, +Goal, -List):  $\approx$  findall, mas os resultados não agrupados por variáveis em Goal que não em Template
3. setof (+Template, +Goal, -List):  $\approx$  bagof, mas os resultados são ordenados e nem repetições

Quantificador Existencial:  $\exists$  - faz ignorar variáveis adicionais em Goal

Se todas as variáveis em Goal mas não em Template estiverem quantificadas existencialmente, então:  
(1) bagof = findall  
(2) setof = findall, sort

| ?- findall(Child, parent(Parent, Child), Children).  
Children = [lisa, bart, maggie, lisa, bart, maggie] ? ;  
no

| ?- bagof(Child, parent(Parent, Child), Children).  
Parent = homer, Children = [lisa, bart, maggie] ? ;  
Parent = marge, Children = [lisa, bart, maggie] ? ;  
no

# Grafos

connected( $X, Y$ ): representa a aresta (dirigida) entre  $X, Y$

DFS: dfs (+  $N_a \rightarrow N_b, +N_f, -\text{Edges}$ )

dfs([ $N_f$ ] -),  $N_f$ , []).

dfs([ $N_a | T$ ],  $N_f$ , [ $E | E_s$ ]) :-

connected( $N_a, N_b, E$ ),  
+ member( $N_b, [N_a | T]$ ),  
dfs( $[N_b, N_a | T], N_f, E_s$ ).

BFS: bfs (+  $\text{queue}, +N_f, -\text{Edges}$ )

bfs([ $[N_f] - E_s | -$ ],  $N_f, E_s$ )

bfs([ $[N_a | T] - E_s | N_a$ ],  $N_f, \text{Yol}$ ) :-  
findall( $[N_b, N_a | T] - [E | E_s]$ ,

(connected( $N_a, N_b, E$ ),  
+ member( $N_b, [N_a | T]$ ))),

append( $N_s, N_s 1, N_s 2$ ),  
bfs( $N_s 2, N_f, \text{Yol}$ ).

Nota: reverse( $E_s \text{Yol}, E_s$ )

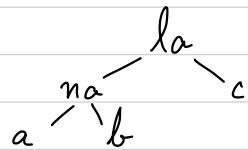
# TP6

5.

a)

b) a la b na c SEM SOLUÇÃO

c) a na b la c



d) a la b ra c SEM SOLUÇÃO

e) a ra b ra c

ra

7.

a. flight ty 1949 from porto to lisbon at 16:15

$$\text{map } (*2) [1, 2, 3] = [2, 4, 6] \rightarrow (\lambda x \rightarrow x * 2)$$

$$\text{zip } [1, a] [3, 4, 5] = [(1, 3), (a, 4)]$$

$$(+3)(-30)(12)$$

$$(10)20=2$$

$$(1)2=0,5$$

$$2: p \text{ with } (+) [1, 2, 3] [1, 2, 3] = [1+1, 2+2, 3+3] = [2, 3, 6]$$

$$\text{filter odd } [1, 2, 3] = [1, 3]$$



$$\text{sqrt } \sqrt{1+2+3} = \text{sqrt } \underbrace{(1+2+3)}_{\text{primero más otro}}$$

$$((+2) \cdot (*3))10 = (+2)((*3)10) = (+2)(10*3) = (3*10)+2 = 32$$

$$(f \circ g)(x) = f(gx) = (f \cdot g)x$$

$$\text{foldl } (-) 0 [1, 2, 3]$$

$$\hookrightarrow (\lambda acc x \rightarrow acc - x)$$

$$[(0-1)-2]-3 = (-1-2)-3$$

$$= -3-3 = -6$$

$$\text{foldr } (-) 0 [1, 2, 3]$$

$$\hookrightarrow (\lambda x acc \rightarrow x - acc)$$

$$1 - [2 - (3 - 0)] = 1 - (2-3)$$

$$= 1 - (-1)$$

$$= 2$$

tipo de função , quanto às folds

recurso , listas em compreensão , funções de ordem

Superiores

map (\*2) (filter (odd) [1, 2, 3, 4])

data = <sup>P</sup><sub>E</sub><sub>A</sub><sub>M</sub> Tipos recursivos ; pattern matching } auxiliam parâmetros  
type = <sup>I</sup><sub>E</sub><sub>N</sub> Minimizar

# HASKELL

Programa Funcional: funções dados  $\xrightarrow{f}$  resultado, nunca modificando variáveis, só aplicando funções

Ex: `main = print (num (map (^2) [1..10]))`  $\rightarrow$  reduções/simplificações

GHCi: read-eval-print-loop (REPL) :l :r :t :g :help

Operadores: +, -, \*, /, ^, div, mod, sqrt, ==, /=, <, >, <=, >=

- argumentos separados por espaços
- aplicação tem maior precedência

$$\begin{aligned} x + y &\equiv (+) x y \\ \text{mod } x 2 &\equiv x \text{ mod' } 2 \end{aligned}$$

Prelúdio - Padrão: head, tail, length, take, drop, ++, reverse, !!, num, product

Ex: `factorial n = product [1..n]` -- comentários {- comentários -}

where: para definições locais (indentação indica o âmbito das declarações)

Ex: `sqrt [] = []`  
`sqrt (x:nxs) = sqrt ns1 ++ [n] ++ sqrt ns2`  
where  $ns1 = [n' | n' < -ns, n' <= k]$   
 $ns2 = [n' | n' < -ns, n' > k]$

Tipo: none, para coleção de valores relacionados, ex:  $\text{Bool}$ ,  $\text{Char}$ ,  $\text{String}$ ,  $\text{Int}$ ,  $\text{Integer}$ ,  $\text{Float}$ ,  $\text{Double}$ ,  $\text{Type}$

Lista: sequência de tamanho variável de elementos do mesmo tipo  
Tuplo: sequência de tamanho fixo de elementos de tipos formalmente dif.

Função A → B: faz corresponder valores tipo A em valores tipo B

Uma função de vários argumentos toma um argumento de cada vez

seta: associa à direita

aplicação: associa à esquerda

Função Polimórfica: admite um tipo com variáveis a  
Sobreposição: permitem tipos restritos =>

Classes: Num, Integral, Fractional, Eq, Ord

from Integral: converte qualquer tipo intérino para qualquer outro tipo numérico

Expressões Condicionais: if ... then ... else ...

Guardas: | ... | ... | otherwise ... testadas por ordem

Equações com Padrões: para distinguir casos - não rejeitam variáveis -

Ex: fst, snd

case ... of ...: equivalente a padrões

Lambda:  $\lambda x \rightarrow \dots$

let ... in ...: define variáveis locais

... : constitui uma lista, elemento a elemento (terminando com lista vazia)  
(n:n)

# Listas

OU  $n : n$

- Coleções de elementos cuja ordem interessa e que podem ser repetidos

Sequências Aritméticas:  $[a .. b]$  ou  $[a, b .. c]$

Listas Infinitas:  $[a ..]$  ou  $[a, b ..]$

- Como não se podem usar ciclos, usa-se recursão
- Ex: product; length; reverse;  $i++$ ; zip; drop

1. Defini o tipo da função
2. Enumera os casos a considerar usando padrões
3. Define o valor nos casos simples
4. Define o valor nos outros casos, assumindo que a função funciona
5. Generaliza e simplifica

Lista em Compreensão:  $[... | \frac{\text{padão}}{\text{gerador}} \leftarrow \text{lista}, \text{condição}] \approx \text{for}$

Ex: concat listas =  $[valor | lista \leftarrow listas, valor \leftarrow lista]$

- Uma string é uma lista de caracteres

divisores  $n = [n | n \in [1..n], n \bmod k == 0]$

testar Primo  $n = \text{divisores } n == [1, n]$

primos  $n = [n | n \in [2..n], \text{testarPrimo } n]$

índices  $n ys = [i | (y, i) \leftarrow \text{zip } ys, [0..(\text{length } ys - 1), i == y]$

fares =  $\text{zip } n : n (\text{tail } n : n)$

fares  $y$  guais =  $\text{length } [(n, n') | (n, n') \leftarrow \text{zip } n : n (\text{tail } n : n), n == n']$

# Funções de Ordenação Inferior

Funções cujo argumento ou resultado é uma função

map: aplica uma função a cada elemento duma lista

$$\text{map } f \text{ vs} = [f n \mid n \in \text{vs}]$$

$$\text{map } f [] = []$$

$$\text{map } f (\text{v}: \text{vs}) = f \text{ v} : \text{map } f \text{ vs}$$

filter: seleciona elementos duma lista que satisfazem um predicado

$$\text{filter } p \text{ vs} = [n \mid n \in \text{vs}, p n]$$

$$\text{filter } p [] = []$$

$$\text{filter } p (\text{v}: \text{vs})$$

$$\begin{cases} p n &= n : \text{filter } p \text{ vs} \\ \text{otherwise} &= \text{filter } p \text{ vs} \end{cases}$$

takeWhile: seleciona o maior prefixo duma lista cujos elementos verificam um predicado

dropWhile: remove o maior prefixo duma lista cujos elementos verificam um predicado

$$\text{takeWhile } p [] = []$$

$$\text{takeWhile } p (\text{v}: \text{vs})$$

$$\begin{cases} p n &= n : \text{takeWhile } p \text{ vs} \\ \text{otherwise} &= [] \end{cases}$$

$$\text{dropWhile } p [] = []$$

$$\text{dropWhile } p (\text{v}: \text{vs})$$

$$\begin{cases} p n &= \text{dropWhile } p \text{ vs} \\ \text{otherwise} &= \text{v}: \text{vs} \end{cases}$$

all: verifica se um predicado é verdadeiro para todos os elementos duma lista

any: verifica se um predicado é verdadeiro para algum dos elementos duma lista

all  $\lambda n : \text{ws} = \text{and} (\text{map } f \text{ ws})$   
all  $\lambda [] = \text{True}$   
all  $\lambda (n:n\Delta) = \lambda n \& \& \text{all } \lambda n : \text{ws}$

any  $\lambda n : \text{ws} = \text{or} (\text{map } f \text{ ws})$   
any  $\lambda [] = \text{False}$   
any  $\lambda (n:n\Delta) = \lambda n || \text{any } \lambda n : \text{ws}$

foldl: transforma uma lista usando uma operação associada à direita  
foldr: transforma uma lista usando uma operação associada à esquerda

foldl  $f g [] = g$   
foldl  $f g (n:n\Delta) = f n (\text{foldl } f g \text{ ws})$

foldr  $f g [] = g$   
foldr  $f g (n:n\Delta) = \text{foldr } f (f g n) \text{ ws}$

$\therefore$  composição de duas funções

repeat  $n = n : n : n : \dots$

repeat  $n = \text{ws}$  where  $\text{ws} = n : \text{ws}$

cycle  $\text{ws} = \text{ws} ++ \text{ws} ++ \text{ws} ++ \dots$

cycle  $\text{ws} = \text{ws}'$  where  $\text{ws}' = \text{ws} ++ \text{ws}'$

iterate  $f n = n : f n : f (f n) : f (f (f n)) : \dots$   
iterate  $f n = n : \text{iterate } f (f n)$

preencher  $n \text{ ws} = \text{take } n (\text{ws} ++ \text{repeat } '')$

aproximações  $q = \text{iterate } (\lambda n \rightarrow 0,5 * (n + q / n)) q$

fibs =  $0 : 1 : \text{zipWith } (+) \text{ fibs} (\text{tail fibs})$

cruzo  $(f : \text{ws}) = \lambda n : \text{cruzo} [n \mid n \leftarrow \text{ws}, n \text{ 'mod' } f \neq 0]$

# Tipos

type: define um novo nome para um tipo existente - sinônimo

data: define novos tipos de dados que podem ser recursivos, enumerando as alternativas de valores do novo tipo

# Módulos

module Foo (..., ..., ...) where

import Foo

# Árvores

listar  $Vazio = []$

listar ( $\text{No } x \text{ esq } di$ ) = listar esq ++ [ $x$ ] ++ listar di

procurar  $x$   $Vazio = \text{False}$

procurar  $x$  ( $\text{No } y \text{ esq } di$ )

|  $x == y = \text{True}$   
 $x < y = \text{procurar } x \text{ esq}$   
 $x > y = \text{procurar } x \text{ di}$

inserir  $x$   $Vazio = \text{No } x \text{ Vazio Vazio}$

inserir  $x$  ( $\text{No } y \text{ esq } di$ )

|  $x == y = \text{No } y \text{ esq } di$   
 $x < y = \text{No } y \text{ (inserir } x \text{ esq) di}$   
 $x > y = \text{No } y \text{ esq } (\text{inserir } x \text{ di})$

# Monads & IO

IO: tipo de comandos que retornam um valor do tipo a

$\text{putChar} :: \text{Chan} \rightarrow \text{IO}()$   
 $(>>) :: \text{IO}() \rightarrow \text{IO}() \rightarrow \text{IO}()$   
 $\text{done} :: \text{IO}()$

$\text{getChar} :: \text{IO Chan}$   
 $\text{return} :: a \rightarrow \text{IO} a$   
 $(>>=) :: \text{IO} a \rightarrow (a \rightarrow \text{IO} b) \rightarrow \text{IO} b$

$\text{putStrLn} :: \text{String} \rightarrow \text{IO}()$   
 $\text{putStrLn} [] = \text{done}$   
 $\text{putStrLn} (x:xs) = \text{putChar } x \gg \text{putStrLn } xs$   
 $\text{putStrLn} = \text{foldr } (\gg) \text{ done . map putChar}$

$\text{getLine} :: \text{IO [Char]}$   
 $\text{getLine} = \text{getChar} \gg= \backslash x \rightarrow$   
if  $x == '\n'$  then  
return []  
else  
 $\text{getLine} \gg= \backslash ws \rightarrow$   
return  $(x:ws)$

$x \leftarrow e_1 \dots \leftrightarrow e \gg= \backslash x \rightarrow \dots$   
 $e_1 \dots \leftrightarrow e \gg \dots$

$\text{echo} :: \text{IO}()$

```

echo = getLine >>= \line =>
if line == "" then
    return ()
else
    putStrLn (map toUpper line) >>
    echo

```

```

echo = do {
    line <- getLine;
    if line == "" then
        return ()
    else do {
        putStrLn (map toUpper line);
        echo
    }
}

```

**Monad:** para  $(*, \mu)$  de um operador associativo  $*$  com um valor de identidade  $\mu$  em que:

$$1. \mu * x = x * \mu = x$$

$$2. (x * y) * z = x * (y * z)$$

**Monad:** para de funções  $(>>=, \text{return})$  em que:

$$1. \text{return } a >>= f = f a$$

$$2. m >>= \text{return } = m$$

$$3. (m >>= f) >>= g = m >>= (\lambda k \rightarrow f k >>= g)$$

Data Maybe a = Nothing | Just a

lookup ::  $\begin{cases} \text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow \text{Maybe } b \\ k((k, v) : \text{assoc}) \\ | \quad k == v = \text{Just } v \\ | \quad \text{otherwise} = \text{lookup } k \text{ assoc} \\ \text{lookup } k [] = \text{Nothing} \end{cases}$

Data Either a b = Left a | Right b

newtype State s a = State ( $s \rightarrow (a, s)$ )

Monads: Identity; Maybe; Error; [] ; IO; State; Reader; Writer; Cont

putChar :: Char -> IO () -- escrever um caracter

putStr :: String -> IO () -- escrever uma cadeia

putStrLn :: String -> IO () -- idem; muda de linha

print :: Show a => a -> IO () -- imprimir um valor

getChar :: IO Char -- ler um caracter

getLine :: IO String -- ler uma linha

getContents :: IO String -- ler toda a entrada padrão

<b>Início</b>	sábado, 4 de dezembro de 2021 às 13:44
<b>Estado</b>	Prova submetida
<b>Data de submissão:</b>	sábado, 4 de dezembro de 2021 às 14:03
<b>Tempo gasto</b>	19 minutos 8 segundos
<b>Nota</b>	6,00 de um máximo de 6,00 (100%)

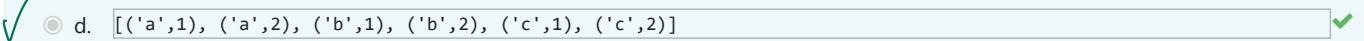
## Pergunta 1

Correta Pontuou 0,500 de 0,500

Qual o resultado de executar o seguinte código?

$[(x,y) \mid x <- \text{"abc"}, y <- [1,2]]$   $\left[ (a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2) \right]$

- a.  $[(\text{'a'}, 1), (\text{'b'}, 1), (\text{'c'}, 1), (\text{'a'}, 2), (\text{'b'}, 1), (\text{'c'}, 2)]$
- b.  $[(\text{'a'}, 1), (\text{'b'}, 2), (\text{'a'}, 1), (\text{'b'}, 2), (\text{'a'}, 1), (\text{'b'}, 2)]$
- c.  $[(\text{'c'}, 1), (\text{'c'}, 2), (\text{'b'}, 1), (\text{'b'}, 2), (\text{'a'}, 1), (\text{'a'}, 2)]$
- d.  $[(\text{'a'}, 1), (\text{'a'}, 2), (\text{'b'}, 1), (\text{'b'}, 2), (\text{'c'}, 1), (\text{'c'}, 2)]$



## Pergunta 2

Qual dos tipos é válido para a seguinte expressão?

$(\text{map } (>0))$   $\left[ \text{Int} \right] \rightarrow \left[ \text{Bool} \right]$

- a.  $[\text{Int}] \rightarrow [\text{Bool}]$
- b.  $[\text{Char}] \rightarrow [\text{Bool}]$
- c.  $(\text{Num a}) \Rightarrow [a] \rightarrow \text{Bool}$
- d.  $(\text{Num a}) \Rightarrow [a] \rightarrow [\text{Bool}]$

## Pergunta 3

Qual o tipo inferido pelo ghci para a seguinte função?

$\text{last} [x] = x$   $\text{last} (x:xs) = \text{last} xs$   $\left[ a \right] \rightarrow a$

- a.  $[a] \rightarrow b$
- b.  $[a] \rightarrow a$
- c.  $(\text{Num a}) \Rightarrow [a] \rightarrow a$
- d.  $[\text{Int}] \rightarrow \text{Int}$

## Pergunta 4

Qual o resultado de executar o seguinte código?

`foldr (+) 7 [1,2,3]`

13

a. [13]

b. 6

c. 7

d. 13



## Pergunta 5

Considere as três afirmações seguintes sobre as diferenças entre "type" e "data":

A - Apenas "data" permite definições recursivas de tipos. ✓

B - Apenas "data" permite definições usando variáveis de tipo (type variables). ↗

C - Apenas "data" define novos padrões para pattern matching. ✓

Qual/Quais destas afirmações estão corretas?

a. Apenas B e C

b. A, B e C

c. Apenas A e B

d. Apenas A e C

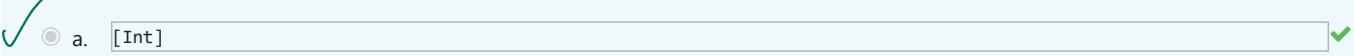


## Pergunta 6

Qual dos tipos é válido para a seguinte expressão?

`head [[1], [2,3], [4,5,6]]`

[Int]



a. [Int]

b. [a]

c. [Char]

d. [String]

## Pergunta 7

Qual o resultado de executar o seguinte código?

`tail (reverse [1,2,3])`

tail ([3,2,1]) = [2,1]



a. [2,3]



b. [2,1]

c. [1,2]

d. 1

## Pergunta 8

Qual o resultado de executar o seguinte código?

`[| (x,y) | (x,y) <- [(1,2),(2,3)], x*y==6]`

$\boxed{[(2,3)]}$

a. `[(2,3)]`



b. `[2,3]`

c. `[(3,3)]`

d. `(2,3)`

## Pergunta 9

Qual o resultado de executar o seguinte código?

`cycle (cycle [1,2,3])`

$\text{cycle} \left( \left[ 1, 2, 3, 1, 2, 3, 1, 2, 3, \dots \right] \right) = \left[ 1, 2, 3, 1, 2, 3, \dots \right]$

Nota: cycle é uma função do Prelúdio-padrão que produz uma lista infinita em que a lista de input é repetida um número infinito de vezes.  
Exemplo:

`cycle [1,2,3] = [1,2,3,1,2,3, ...]`

a. `[1,1,1,1,1,1 ...]`

b. O cálculo da expressão gera um erro.

c. `[1,2,3,1,2,3, ...]`



d. `[1,1,2,2,3,3,3,1,1,2,2,3,3, ...]`

## Pergunta 10

Qual o resultado de executar o seguinte código?

`head (zip [1..10] (tail [1..10]))`

$\text{head} \left( \text{zip} \left[ 1..10 \right] \left[ 2..9 \right] \right) = (1,2)$

a. `(1,10)`

b. `(1,1)`

c. `(2,1)`

d. `(1,2)`



## Pergunta 11

$\text{data Arv } a = \text{Folha } | \text{ No } a (\text{Arv } a) (\text{Arv } a)$

Qual das seguintes alternativas define um tipo de árvores binárias com anotações apenas nos nós?

a. `data Arv a = Folha | No a (Arv a) (Arv a)`



b. `data Arv a = Folha a | No a (Arv a) (Arv a)`

c. `data Arv a = Folha a | No (Arv a) (Arv a)`

d. `data Arv = Folha | No Arv Arv`

## Pergunta 12

Qual o resultado de executar o seguinte código?

```
filter (/='a') "abba" lelo
```

a. "ab"

b. ""

c. "bb"

d. "aa"



**Início** quinta, 27 de outubro de 2022 às 17:14

**Estado** Prova submetida

**Data de** quinta, 27 de outubro de 2022 às 17:33

**submissão:**

**Tempo gasto** 19 minutos 5 segundos

**Nota** 4,75 de um máximo de 6,00 (79%)

## Pergunta 1

Correta Pontuou 0,500 de 0,500

What is the type of the following function?

```
orderedPair (a, b)
| a <= b = (a, b)
| otherwise = (b, a)
```

$$(\text{Ord } a) \Rightarrow (a, a) \rightarrow (a, a)$$

- a.  $(\text{Num } a, \text{ Num } b) \Rightarrow (a, b) \rightarrow (b, a)$
- b.  $(\text{Ord } a) \Rightarrow (a, a) \rightarrow (a, a)$
- c.  $(\text{Num } a, \text{ Num } b) \Rightarrow (a, b) \rightarrow (a, b)$
- d.  $(\text{Ord } a, \text{ Ord } b) \Rightarrow (a, b) \rightarrow (b, a)$
- e.  $(\text{Num } a, \text{ Ord } a, \text{ Num } b, \text{ Ord } b) \Rightarrow (a, b) \rightarrow (b, a)$

## Pergunta 2

Correta Pontuou 0,500 de 0,500

What is the result of the following expression?

```
(length . (filter (> 0))) [1, 2, -3, 4, -5]
```

$$\text{length} [1, 2, 4] = 3$$

- a. 0
- b. The evaluation of the expression produces an error.
- c. 3
- d. 1
- e. 2

## Pergunta 3

Correta Pontuou 0,500 de 0,500

$$(\text{Num } a) \Rightarrow [[a] \rightarrow [a]]$$

What is the type of the following expression?

```
[(+++) [], map (+1)]
```

- a.  $[[a] \rightarrow [a]]$
- b.  $(\text{Num } a) \Rightarrow [a]$
- c.  $(\text{Num } a, \text{ Num } b) \Rightarrow [[a] \rightarrow [b]]$
- d.  $[[a] \rightarrow [b]]$
- e.  $(\text{Num } a) \Rightarrow [[a] \rightarrow [a]]$

## Pergunta 4

Correta Pontuou 0,500 de 0,500

What is the type of the following function?

```
fun (x, y, _) = (y, x, y)
fun (_, y, x) = (y, x, y)
```

$$(\alpha, \beta, \alpha) \rightarrow (\beta, \alpha, \beta)$$

- a.  $(\alpha, \alpha, \alpha) \rightarrow (\alpha, \alpha, \alpha)$
- b.  $(\alpha, \beta, \gamma) \rightarrow (\alpha, \beta, \gamma)$
- c.  $(\alpha, \beta, \alpha) \rightarrow (\beta, \alpha, \beta)$
- d.  $(\alpha, \beta, \gamma) \rightarrow (\delta, \epsilon, \zeta)$
- e.  $(\alpha, \alpha, \alpha) \rightarrow (\beta, \beta, \beta)$

## Pergunta 5

Correta Pontuou 0,500 de 0,500

What is the result of the following expression?

```
[(a, b) | a <- "abc", b <- [1, 2], a <= 'd']
```

$$[(\alpha, 1), (\alpha, 2), (\beta, 1), (\beta, 2), (\gamma, 1), (\gamma, 2)]$$

- a.  $[('a', 1), ('b', 1), ('c', 1), ('a', 2), ('b', 2), ('c', 2)]$
- b.  $[('a', 1), ('a', 2), ('b', 1), ('b', 2), ('c', 1), ('c', 2)]$
- c. The evaluation of the expression produces an error.
- d.  $[('a', 1), ('b', 1), ('c', 1)]$
- e.  $[]$

## Pergunta 6

Correta Pontuou 0,500 de 0,500

What is the result of the following expression?

```
foldl (/) 200 [1, 2, 4]
```

$$200 / 1 = 200 / 2 - 100 / 4 = 25$$

- a.  $50.0$
- b. The evaluation of the expression produces an error.
- c.  $25.0$
- d.  $100.0$
- e.  $1.0e-2$

## Pergunta 7

Correta Pontuou 0,500 de 0,500

Which of the following Prelude functions does NOT necessarily return a list?

!!

- a. `(++)`
- b. `(::)`
- c. `zip`
- d. `init`
- e. `(!!)`

✓

✓

## Pergunta 8

Correta Pontuou 0,500 de 0,500

Consider the three following statements about the "type" and "data" keywords.

- A - "type" does not allow the use of type variables, unlike "data". 
- B - "type" does not allow recursive type definitions. 
- C - It is possible to define an instance of Eq using "data". 

Which statements are correct?

- ✓
- a. Only B and C.
  - b. Only A and B.
  - c. Only A and C.
  - d. Only B.
  - e. A, B and C.

✓

## Pergunta 9

Incorreta Pontuou -0,125 de 0,500

Among the types Maybe, State and IO, which of them are monads?

TODOS

- ✓
- a. Maybe, State and IO.
  - b. Only Maybe and IO. 
  - c. Only State and IO.
  - d. Only IO.
  - e. None of these types is a monad.

## Pergunta 10

Correta Pontuou 0,500 de 0,500

What is the correct type of the following function?

```
howdy name = putStrLn ("howdy " ++ name ++ "!")
```

*String → IO ()*

- a. `String -> IO ()`
- b. `String -> String`
- c. `IO ()`
- d. `IO (String)`
- e. `String -> IO (String)`

## Pergunta 11

Incorreta Pontuou -0,125 de 0,500

Haskell has lazy evaluation, which allows for ...

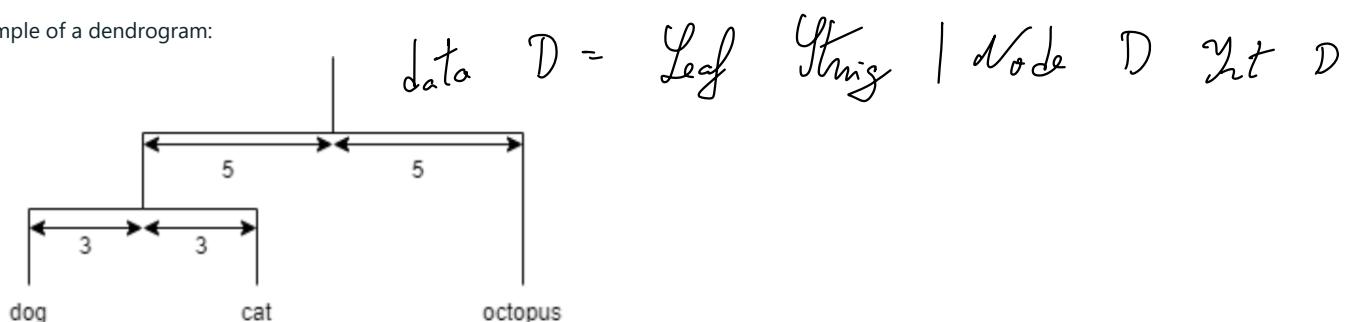
- a. the automatic inference of the functions' type.
- b. the optimization of the memory consumption of a program, in exchange for an increased execution time.
- c. the definition of polymorphic functions.
- d. certain computations with infinite data structures to be finite.
- e. the definition of higher-order functions.

## Pergunta 12

Correta Pontuou 0,500 de 0,500

Consider a dendrogram as a binary tree where each path leads to a string. Each non-leaf node of the dendrogram specifies the horizontal distance from the father node to each of the two child nodes. A father node is always at an equal horizontal distance from both its children.

Example of a dendrogram:



What is the most correct definition of the Dendrogram type?

- a. `data Dendrogram = Leaf String | Node Int Int Dendrogram`
- b. `(Integral a) => data Dendrogram = Leaf (String, a) | Node Dendrogram Dendrogram`
- c. `data Dendrogram = Leaf (String, Int) | Node Dendrogram Dendrogram`
- d. `(Integral a) => data Dendrogram = Leaf String | Node Dendrogram a a Dendrogram`
- e. `data Dendrogram = Leaf String | Node Dendrogram Int Dendrogram`