# Fundamentos de Algoritmos

CIC/UFRGS

2006/1

# Mais em definições auto-referenciáveis: roteiro

- Estruturas em estruturas: exemplo das árvores de hereditariedade
- Árvores binárias de pesquisa
- Listas de listas
- Avaliações em Scheme

# Estruturas em estruturas

- árvore de hereditariedade
- cada indivíduo é um elemento da árvore
- um “filho” é conectado à “mãe” e ao "pai"na árvore
- árvore de ancestrais: dado qualquer nó, encontram-se os ancestrais mas não os descendentes

# Estrutura de cada indivíduo

- (**define-struct** *filho* (pai mãe nome data olhos))

- Uma *filho* é uma estrutura: (make-filho $p\ m\ n\ d\ o$) onde
  $p$ e $m$ são estruturas do tipo *filho*
  $n$ e $o$ são símbolos
  $d$ é número.

- estruturas auto-ferenciáveis devem ter pelo menos duas cláusulas: *mãe* e *pai*, em determinadas situações, podem não ser do tipo *filho*!

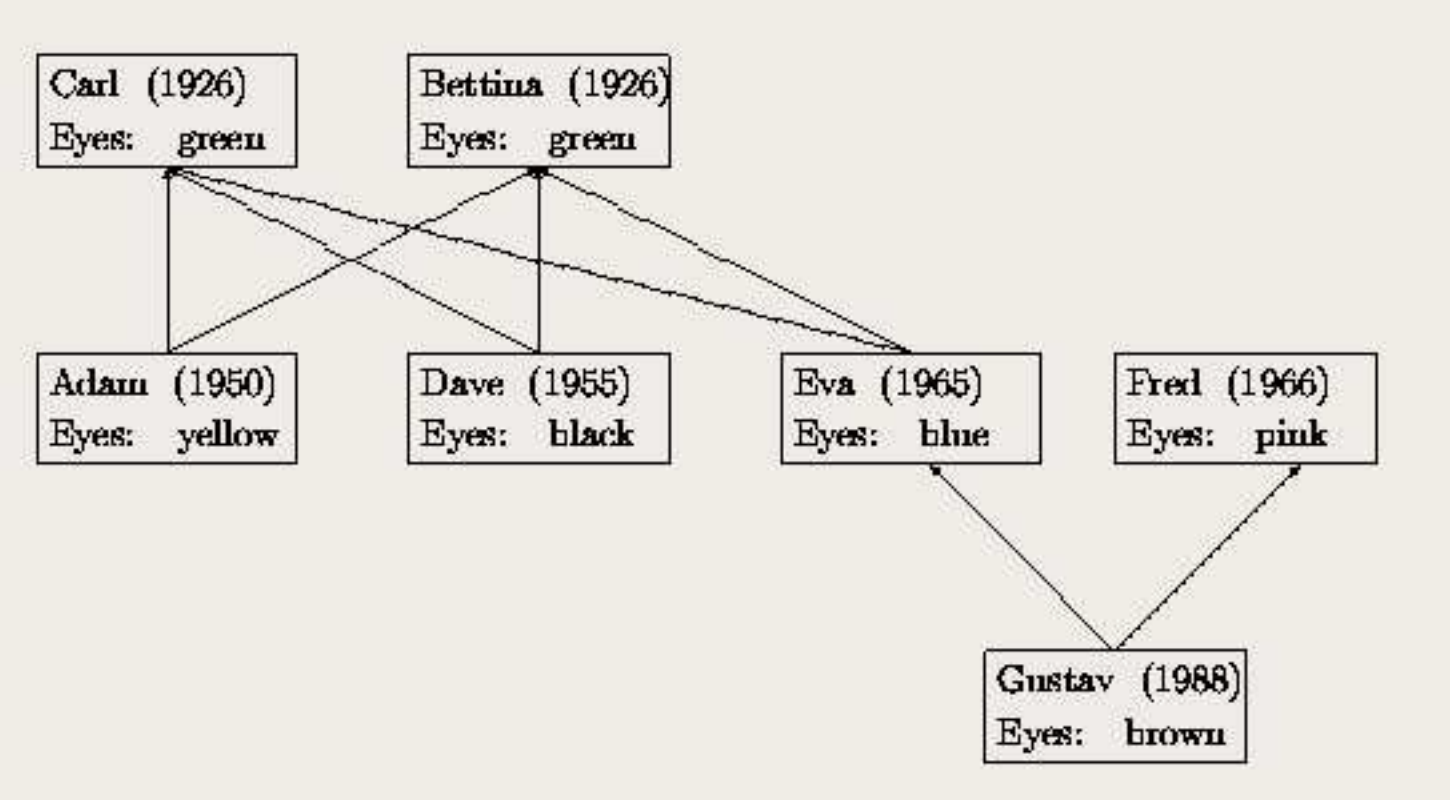


Figura 1: Árvore de hereditariedade: tipo ascendente

# Criação de uma estrutura para cada indivíduo

- (make-filho Carl Bettina 'Adam 1950 'amarelos)
- (make-filho empty empty 'Bettina 1926 'verdes)
- Um filho é uma estrutura

  (make-filho $p\ m\ n\ d\ o$) onde

  $p$ e $m$ são estruturas *empty* ou do tipo *filho*
  $n$ e $o$ são símbolos
  $d$ é número

# Elementos da árvore

- Um nó da árvore é:
  - *empty* ou
  - (make-filho $p$ $m$ $n$ $d$ $o$) onde $p$ e $m$ são nós, $n$ e $o$ são símbolos e $d$ é número.

- Exemplo: geração do nó Adam

```
(make-filho (make-filho empty empty 'Carl 1926 'verdes)
            (make-filho empty empty 'Bettina 1926 'verdes)
            'Adam
            1950
            'amarelos)
```

# Definição de variável para cada nó

```
;; Geração antiga:
(define Carl (make-filho empty empty 'Carl 1926 'verdes))
(define Bettina (make-filho empty empty 'Bettina 1926 'verde

;; Geração intermediária:
(define Adam (make-filho Carl Bettina 'Adam 1950 'amarelos))
(define Eva (make-filho Carl Bettina 'Eva 1965 'azuis))
(define Fred (make-filho empty empty 'Fred 1966 'pink))

;; Geração nova:
(define Gustav (make-filho Fred Eva 'Gustav 1988 'castanhos)
```

# Template para funções sobre a árvore de hereditariedade

```
;; função-x : árvore de hereditariedade -> ???
(define (função-x arv)
   (cond
      [(empty? arv) ... ]
      [else
         ...(função-x (filho-pai arv))...
         ...(função-x (filho-mãe arv))...
         ...(filho-nome arv)...
         ...(filho-data arv)...
         ...(filho-olhos arv)... ]))
```

# Ancestrais de Olhos Azuis

```
;; ancestral-olhos-azuis? nó -> boolean
;; Objetivo: determina se um nó da árvore possui uma estrutu
;; filho com olhos azuis
(define (ancestral-olhos-azuis? nó)...)
```

- (ancestral-olhos-azuis? Carl)
- (ancestral-olhos-azuis? Gustav)
- (ancestral-olhos-azuis? (filho-mãe nó))
- (ancestral-olhos-azuis? (filho-pai nó))

# Ancestrais de Olhos Azuis: versão com cond

```
;; ancestral-olhos-azuis? nó -> boolean
;; determina se um nó da árvore possui uma estrutura filho
;; com olhos azuis
(define (ancestral-olhos-azuis? nó)
   (cond
     [(empty? nó) false]
     [else (cond
       [(symbol=? (filho-olhos nó) 'azuis) true]
       [(ancestral-olhos-azuis? (filho-pai nó)) true]
       [(ancestral-olhos-azuis? (filho-mãe nó)) true]
       [else false])
     ]))
```

# Ancestrais de Olhos Azuis: versão com or

```
;; ancestral-olhos-azuis? nó -> boolean
;; determina se um nó da árvore possui uma estrutura filho
;; com olhos azuis
(define (ancestral-olhos-azuis? nó)
   (cond
     [(empty? nó) false]
     [else (or (symbol=? (filho-olhos nó) 'azuis)
               (or (ancestral-olhos-azuis? (filho-pai nó))
                   (ancestral-olhos-azuis? (filho-mãe nó))
                )
           )
     ]))
```

# Exercícios

- faça testes de mesa para a função *ancestral-olhos-azuis?* considerando cada nó da árvore.

- 14.1.3 Desenvolva a função `conte-pessoas`. A função recebe um nó de uma árvore de hereditariedade e devolve o número de ancestrais com informação na árvore.

- 14.1.5 Desenvolva a função `cor-dos-olhos`, o qual consome um nó de uma árvore de hereditariedade e produz uma lista de todas as cores de olhos de seus ancestrais. Uma cor pode ocorrer mais de uma vez na lista.

# Árvores

- Existe um nó denominado raiz da árvore
- A raiz de uma árvore é chamada de pai de suas subárvores
- Nós com o mesmo nó pai são denominados irmãos
- O número de subárvores de um nó é por definição o grau do nó
- Grau da árvore é o grau máximo entre todos os nós
- Um nó sem subárvores é denominado uma folha da árvore, ou seja, um nó com grau=0
- O comprimento de um caminho desde a raiz $R$ até um nó $N$ denomina-se o nível do nó $N$
- O maior nível de uma árvore é denominado a altura ou profundidade da árvore

# Árvores Binárias

- árvores são estruturas bem conhecidas dos programadores
- possui a estrutura *nó* em vez de *filho*
- um nó de uma árvore binária tem a seguinte estrutura:
  (**define-struct** nó ($id\ nome\ esq\ dir$))

# Árvores Binárias

- Exemplos de árvores binárias:
  - (make-nó 15 'd false (make-nó 24 'i false false))
  - (make-nó 15 'd (make-nó 87 'h false false) false)
  - (make-nó 24 'd false false)

# Exemplos de Árvores Binárias

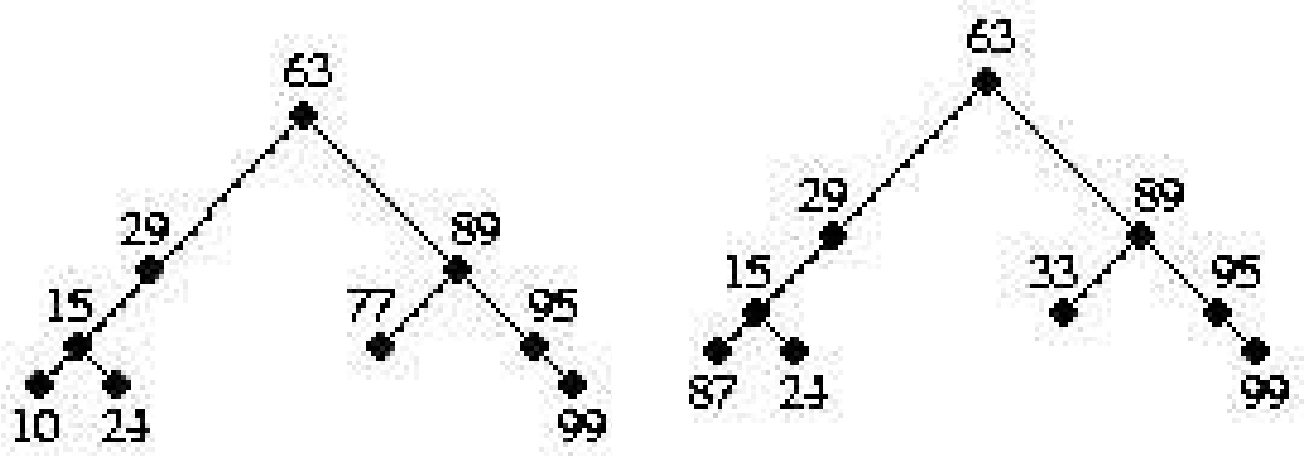


Figura 2: Dir: árvore binária de pesquisa e Esq: árvore binária

# Árvores Binárias de Pesquisa

- Uma AB que possui uma sequência ordenada da informação é chamada de `árvore binária de pesquisa - ABP`

- uma árvore binária de pesquisa é usada para armazenar/recuperar informação

- uma árvore binária (AB) é:
  - false (também poderia ser *empty*) ou
  - (make-nó $v$ $n$ $e$ $d$) onde
    $v$ é um número, $n$ é um símbolo, $e$ e $d$ são árvores binárias.

# Exercícios

- Desenhe árvores binárias de dimensões (número de nós) variadas
- 14.2.1 Desenvolva a função *pesquisa-AB*. A função recebe um número $n$ e uma AB. Se a árvore contém um nó com o campo *id* igual a $n$, a função retorna o *nome* do nó. Caso contrário, a função retorna *false*.

# Árvore Binária de Pesquisa

Uma árvore binária de pesquisa é:

- false é sempre uma AB
- (make-nó $val\ nome\ esq\ dir$) é uma ABP se:
  1. $esq$ e $dir$ são ABP
  2. todos $val$ dos nós $esq$ de um nó são menores que o $val$ deste nó
  3. todos $val$ dos nós $dir$ de um nó são maiores que o $val$ deste nó

# Recuperação de Informação em uma ABP

- encontrar um nó em uma ABP é mais fácil que em um AB
  - em uma AB todos os nós devem ser verificados
  - em uma ABP apenas parte dos nós devem ser verificados: somente o ramo da esquerda o direita

# Construção uma ABP

- construir uma AB é mais fácil que construir uma ABP

- a construção de uma ABP pode ser feita com o uso de duas funções
  - a função insere-nó adiciona um elemento na árvore.
  - a função constrói-ABP adiciona na árvore um a um os nós de uma lista

# Exercícios

- 14.2.3 Desenvolva a função *em-ordem* que recebe uma árvore binária e produz uma lista de todos os *id* desta árvore. A lista contém os números da esquerda para direita. OBS: Use a operação append do scheme para concatenar listas (append (1 2 3) (5 6 7)) = (1 2 3 5 6 7).

- 14.2.4 Desenvolva a função *pesquisa-ABP*. A função recebe um número $n$ e uma ABP. Se existe um nó com o campo *id* igual a $n$ a função retorna seu nome, caso contrário retorna *false*.

# Exercícios

- 14.2.5 Desenvolva a função *insere-nó*. A função recebe uma ABP $B$, um número $n$ e um símbolo $s$. A função cria uma nova ABP igual a anterior mas com a adição do nó com valores $n$ e $s$.

- 14.2.6 Desenvolva a função constrói-ABP a partir de uma lista-de-números. Use as listas exemplos apresentadas na página 204 do livro.
  - Uma lista-de-números é:
    * empty
    * (cons (list $id$ $nome$) $lnum$) onde $id$ é um número, $nome$ é um símbolo, e $lnum$ é uma lista de números

# Função insere-nó

OBS: No DrScheme, escolher Language=Intermediate Student

(**define-struct** nó ($id\ nome\ esq\ dir$))

```
;; insere-nó : ABP numbero símbolo -> ABP
;; cria uma nova árvore ABP igual a ABP passada na entrada, ma
;; com a adição do nó com os valores n e s passados na entrada
(define (insere-nó abp n s)...)
```

# Função insere-nó

```
(define (insere-nó abp n s)
  (cond
    [(eq? abp false) (make-node n s false false)]
    [else
     (cond
       [(< n (node-id abp))
        (make-node (node-id abp)
                   (node-nome abp)
                   (insere-nó (node-esq abp) n s)
                   (node-dir abp))]
       [(> n (node-id abp))
        (make-node (node-id abp)
                   (node-nome abp)
                   (node-esq abp)
                   (insere-nó (node-dir abp) n s))]
       [else (error 'insere-nó "Id já inserido")])]))
```

# Função insere-nó: Testes

```
(equal? (insere-nó false 6 'b)  (make-node 6 'b false false))
(equal? (insere-nó (make-node 4 'a false false) 5 'a)
        (make-node 4 'a false (make-node 5 'a false false)))
(equal? (insere-nó (make-node 4 'a false false) 3 'g)
        (make-node 4 'a (make-node 3 'g false false) false))
(equal? (insere-nó
    (make-node 4 'a (make-node 2 'a false false) false) 3 'g)
    (make-node 4 'a
               (make-node 2 'a false
               (make-node 3 'g false false)) false)
)
```

# Função constrói-ABP

```
;; constrói-ABP : lista de números -> ABP
;; constrói uma ABP com os números informados por uma lista
;; os números possuem um símbolo associado
(define (constrói-ABP lst)
  (cond
    [(empty? lst) false]
    [else
       (insere-nó
         (constrói-ABP (rest lst))
         (first (first lst))
         (second (first lst))
       )
    ]))
```

# Testes

```
(equal? (constrói-ABP '()) false)
(equal?
 (constrói-ABP '((1 a) (18 b) (2 g)))
 (make-node 2 'g
            (make-node 1 'a false false)
            (make-node 18 'b false false)
 ))
```

# Listas em listas

- uma página web possui links a outras páginas web
- cada página pode conter um número indeterminado de links a outras páginas web
- grafo das páginas web: cada página é um nó
- árvore vs. grafo

# Grafo das páginas web

- Uma página web é:

  1. empty
  2. (cons s wp) onde $s$ é um símbolo e $wp$ é uma página web
  3. (cons $ewp$ $wp$) onde $ewp$ e $wp$ são páginas web.

- A definição de dados acima tem:

  – três cláusulas (em vez de duas)
  – três auto-referências (em vez de uma)

# Exemplos de páginas web

- Página1:

  '(O projeto TeachSchme! objetiva melhorar a habilidade para organização e resolução de problemas de estudantes. O projeto provê software e notas, assim como exercícios e soluções para os professores.)

- Página2:

  '(A página web TeachScheme. Aqui você pode encontrar:
  (Notas para professores)
  (Tutorial para (DrScheme: um ambiente de programação))
  (Exercícios)
  (Soluções para os Exercícios)
  Para informação adicional: escreva para scheme@cs)

# Conta símbolos de uma página web

```
;; conta-símbolos: wp -> número
;; conta o número de símbolos de uma página web
(define (conta-símbolos a-wp)
   (cond
      [(empty? a-wp)...]
      [(symbol? (first a-wp))
            ... (first a-wp)
            ... (conta-símbolos (rest a-wp)) ...]
      [else ... (conta-símbolos (first a-wp))
            ... (conta-símbolos (rest a-wp)) ...]
   )
)
```

# Conta símbolos de uma página web

```
;; conta-símbolos: wp -> número
;; conta o número de símbolos de uma página web
(define (conta-símbolos a-wp)
   (cond
      [(empty? a-wp) 0]
      [(symbol? (first a-wp))
           (+ 1 (conta-símbolos (rest a-wp)))]
      [else
           (+ (conta-símbolos (first a-wp))
              (conta-símbolos (rest a-wp))
           )
      ]
   ))
```

# Exercícios

- 14.3.2 Desenvolva a função *ocorre*. A função recebe uma página web e um símbolo $s$. A função produz o número de vezes que o símbolo $s$ ocorre na página web, ignorando as páginas embutidas.

- 14.3.3 Desenvolva a função *troca*. A função recebe dois símbolos, *novo* e *velho*, e uma página web $pw$. A função produz uma página estruturada similarmente a $wp$ mas com todas ocorrências de $velho$ trocadas por $novo$.

# Avaliando Expressões Scheme

- Para representação das operações de adição e multiplicação:

  (define-struct soma (dir esq))

  (define-struct mult (dir esq))

- Exemplos: 3 $\rightarrow$ 3

  (* 3 10) $\rightarrow$ (make-mult 3 10)

  (+ (* x x)(* y y)) $\rightarrow$ (make-add (make-mult 'x 'x)(make-mult 'y 'y))

- Exercício 14.4.3 Apresenta uma definição de dados para expressões numéricas. Desenvolva a função *avalie-expressão*. A função recebe um expressão numérica e calcula o resultado.