Definições de Dados Auto-Referenciáveis

Fundamentos de Algoritmos

INF05008

Estruturas em Estruturas

- **Exemplo:** Árvore genealógica
- Cada indivíduo é um elemento da árvore (**nó**)
- 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 Um Elemento

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

- ***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 um número.

- Estruturas auto-referenciáveis devem ter pelo menos **duas cláusulas**: *mãe* e *pai* (em determinadas situações, podem não ser do tipo *filho*)

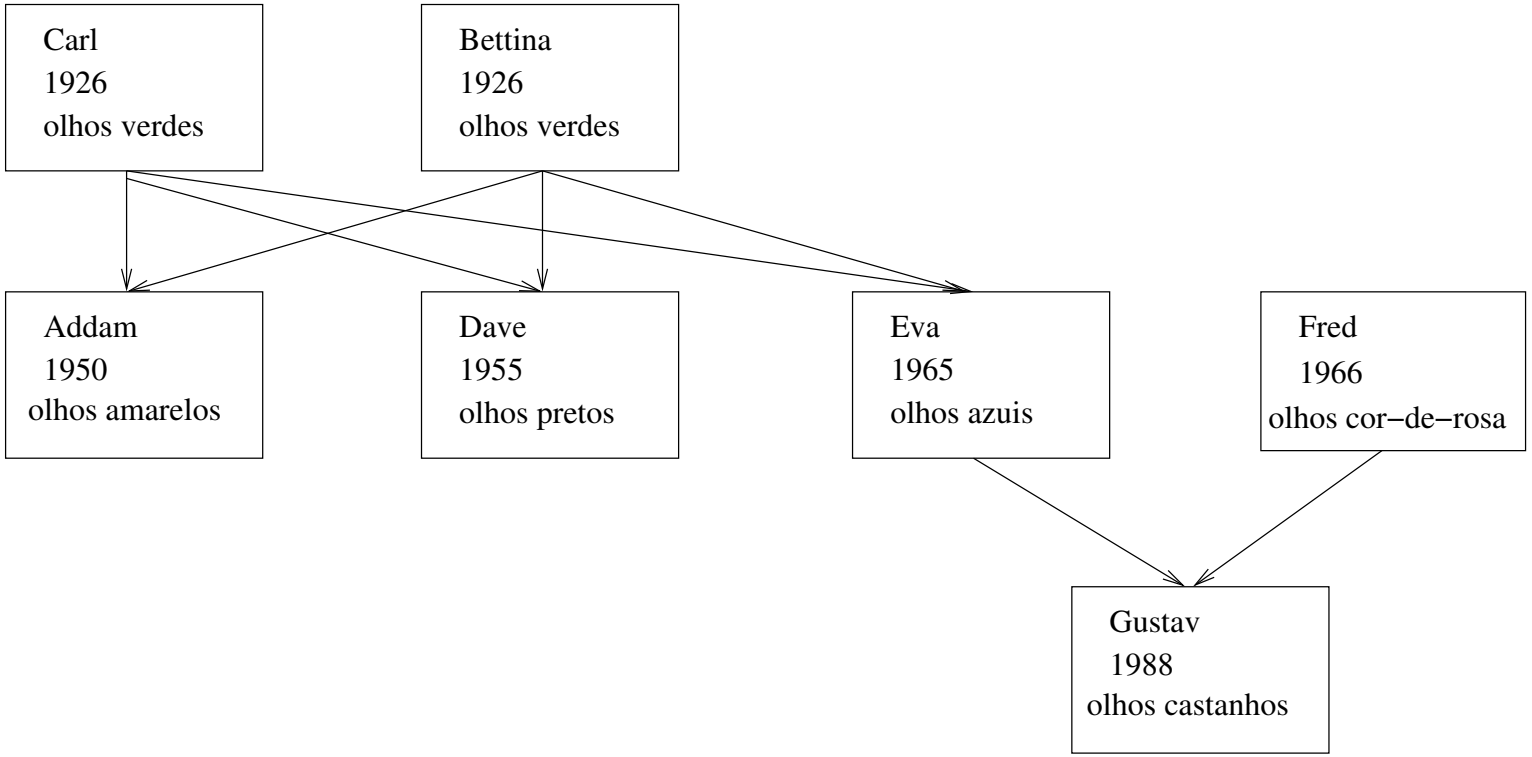

Figura 1: Árvore genealógica do tipo ascendente

Criação de Estrutura para um Elemento

- Quando os pais são conhecidos, usa-se uma **referência** a eles:
  `(make-filho Carl Bettina 'Adam 1950 'amarelos)`

- Quando os pais não são conhecidos, usamos `empty` para representar uma **referência nula**:
  `(make-filho empty empty 'Bettina 1926 'verdes)`

- Refazendo a definição anterior, *filho* é uma estrutura

  `(make-filho` p m n d o`)`, onde

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

Elementos da Árvore

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

Associação de Nomes aos Nós

- Criação da árvore dessa forma requereria **repetições de dados**
- Por isso, utilizamos **referências às estruturas** em vez de recriar estruturas de forma redundante
- Assim, associamos **nomes** às estruturas e usamos estes nomes para nos referirmos a elas

Associação de Nomes aos Nós (cont.)

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

;; 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 Genealógica

```
;; função-x : árvore-genealógica -> ???

(define (função-x arv)
   (cond
      [(empty? arv) ... ]
      [else
         ...(função-x (filho-pai arv))...
         ...(função-x (filho-mae arv))...
         ...(filho-nome arv)...
         ...(filho-data arv)...
         ...(filho-olhos arv)... ]))
```

Ancestrais de Olhos Azuis

```
;; ancestral-olhos-azuis? : nó -> boolean
;; Determina se em uma dada árvore de ascendência
;; existe alguma uma estrutura filho com olhos azuis

(define (ancestral-olhos-azuis? nó) ... )

;; Exemplos:

(ancestral-olhos-azuis? Carl) produz false

(ancestral-olhos-azuis? Gustav) produz true
```

Ancestrais de Olhos Azuis: versão com cond

```
;; ancestral-olhos-azuis? nó -> boolean
;; Determina se em uma dada árvore de ascendência
;; existe alguma 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-mae 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))
                )
               )
     ]))
```

Árvores

- Existe um nó denominado raiz da árvore
- A raiz de uma árvore é chamada de pai de suas **sub-árvores**
- Nós com o mesmo nó-pai são denominados irmãos
- O número de sub-árvores de um nó é, por definição, o grau do nó
- Grau da árvore é o grau máximo entre todos os nós
- Um nó sem sub-árvores 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
- Possuem 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 (cont.)

- Exemplos de árvores binárias:

```
(make-nó 15 'd empty (make-nó 24 'i empty empty))
(make-nó 15 'd (make-nó 87 'h empty empty) empty)
(make-nó 24 'd  empty empty)
```

Árvores Binárias de Pesquisa

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

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

- Uma ABP é usada para armazenar/recuperar informações

Árvore Binária de Pesquisa

Uma árvore binária de pesquisa é:

- `empty` 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

Exemplos de Árvores Binárias

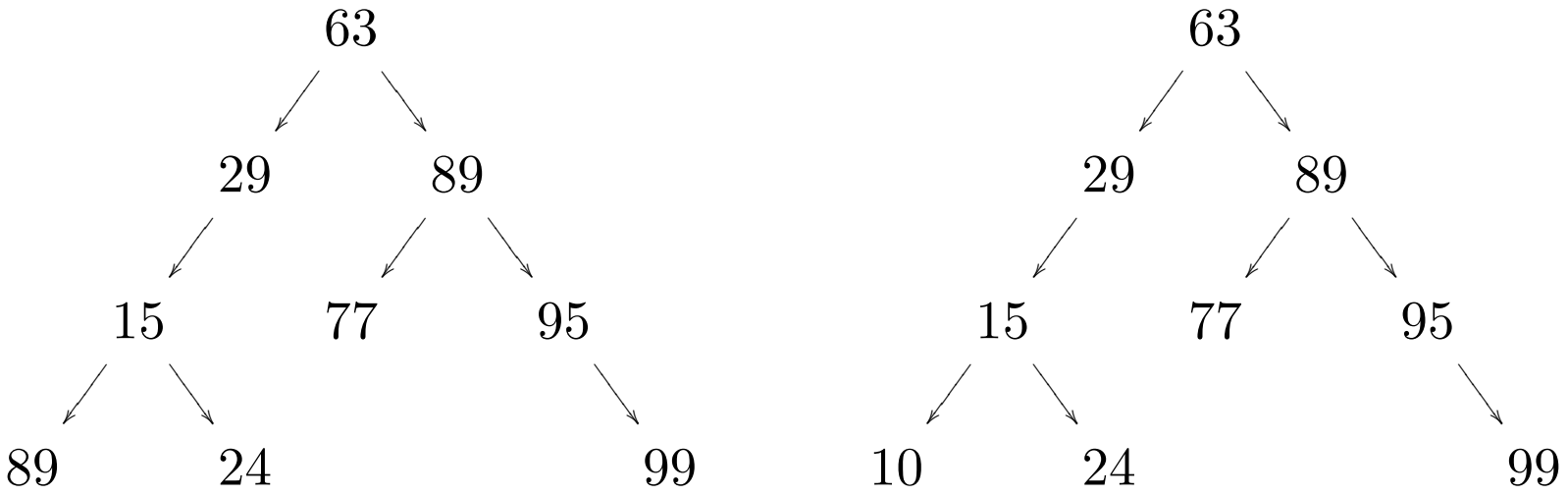

Figura 2: Árvore binária (esq) e árvore binária de pesquisa (dir)

Recuperação de Informação em uma ABP

- **Encontrar um nó** em uma ABP é mais fácil que em uma AB
  - Em uma AB, **todos os nós devem ser verificados**
  - Em uma ABP, apenas parte dos nós devem ser verificados: **somente a sub-árvore da esquerda ou da direita** de cada nó

Construção de uma ABP

- **Construir** uma AB é mais fácil do 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 à árvore
  - A função **`constrói-ABP`** adiciona à árvore, um a um, os elementos de uma lista

Função `insere-nó`

```
(define-struct nó (id nome esq dir))
;; empty, ou
;; (make-nó id n e d), onde 'id' é número,
;; 'n' é símbolo e 'e' e 'd' são nós

;; insere-nó : ABP número símbolo -> ABP
;; Cria uma nova árvore ABP igual à ABP passada na entrada,
;; mas com a adição do nó com os valores 'id' e 'n' passados
;; na entrada.

(define (insere-nó abp id n)...)
```

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

Função `insere-nó`: Testes

```
(equal? (insere-nó empty 6 'b)  (make-nó 6 'b empty empty))

(equal? (insere-nó (make-nó 4 'a empty empty) 5 'a)
        (make-nó 4 'a empty (make-nó 5 'a empty empty)))

(equal? (insere-nó (make-nó 4 'a empty empty) 3 'g)
        ??)

(equal? (insere-nó
    (make-nó 4 'a (make-nó 2 'a empty empty) empty) 3 'g)
    ??)
```

Função constrói-ABP

```
;; constrói-ABP : lista-de-números -> ABP
;; Constrói uma ABP a partir de números informados
;; em uma lista, os quais possuem um símbolo associado

(define (constrói-ABP ldn)
  (cond
    [(empty? ldn) empty]
    [else
       (insere-nó
         (constrói-ABP (rest ldn))
         (first (first ldn))
         (second (first ldn))
       )
    ]))
```

Testes

```
(equal? (constrói-ABP (list )) empty)

(equal?
 (constrói-ABP (list (list 1 'a) (list 18 'b) (list 2 'g)))
 (make-nó 2 'g
          (make-nó 1 'a empty empty)
          (make-nó 18 'b empty empty)
 ))

(equal? (constrói-ABP (list (list 4 'a) (list 6 'b)
                            (list 1 'c) (list 8 'h)
                            (list 9 'x) (list 5 'l))
        ??)
```

Exercício

- Crie uma função `gera-lista`, a qual recebe uma ABP e cria uma lista com os `id` dos nós da árvore.

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**

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 TeachScheme! 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))
           )
      ]
   ))
```

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) → (make-mult 3 10)

(+ (* x x)(* y y)) → (make-add (make-mult 'x 'x)(make-mult
'y 'y))
```

Exercício

- Apresente uma definição de dados para expressões numéricas. Desenvolva a função `avalie-expressão`. A função recebe uma expressão numérica e calcula o seu resultado.