TIPOS ABSTRATOS DE DADOS
¾ DEFINIÇÃO E EXEMPLOS ¾

Bruno Maffeo
Departamento de Informática
PUC-Rio

TERMINOLOGIA BÁSICA

ALGORITMO

Um **algoritmo** pode ser visto como uma seqüência de ações expressas em termos de uma linguagem de programação, constituindo parte da solução de um tipo determinado de problema. Ou seja, um algoritmo corresponde à descrição do padrão de comportamento associado aos **elementos funcionais ativos** de um processamento e deve ser expresso em termos de um conjunto finito de ações especificadas por meio de uma linguagem de programação.

ESTRUTURAS DE DADOS

Na linguagem de programação adotada, **estruturas de dados** dão suporte à descrição dos **elementos funcionais passivos** do padrão de comportamento acima referido, complementando o **algoritmo** que constitui parte da solução do problema considerado. Ambos, algoritmo e estruturas de dados, compõem o **programa** a ser executado pelo computador.

Não é útil estudar estruturas de dados sem considerar os algoritmos básicos a elas associados. Da mesma forma, a escolha de um algoritmo depende, em geral, da estrutura de dados associada.

PROGRAMA

Programar é, basicamente, estruturar dados e construir algoritmos.

Um **programa** é uma formulação concreta, em termos de uma linguagem de programação, de um **procedimento** abstrato que atua sobre um **modelo de dados** também abstrato. Ambos, procedimento e modelo de dados, são representações abstratas de algoritmo e estruturas de dados, respectivamente, e **não** devem ser expressos em termos de uma linguagem de programação.

REPRESENTAÇÕES DE DADOS

TIPOS DE DADOS

Em uma linguagem de programação, é importante **classificar** constantes, variáveis e valores gerados por expressões/funções de acordo com o seu **tipo de dados**. Um tipo de dados deve caracterizar o conjunto de valores a que uma constante pertence ou o conjunto de valores que pode ser assumido por uma variável ou gerado por uma expressão/função.

Um tipo de dados **elementar** (ou **simples**) é caracterizado por um conjunto — **domínio** — de valores indivisíveis, tais como os definidos pelos tipos *numeral*, *texto* e *símbolo* da linguagem de programação Scheme.

Um tipo de dados **estruturado** (ou **complexo**) define, em geral, uma coleção homogênea (de mesmo tipo) de valores elementares/estruturados **ou** um agregado de valores de tipos diferentes. Um exemplo é o tipo *lista* de Scheme.

TIPO ABSTRATO DE DADOS

Abstraída qualquer linguagem de programação, um **tipo abstrato de dados** (*TAD*) pode ser visto como um modelo matemático que encapsula um **modelo de dados** e um conjunto de **procedimentos** que atuam com exclusividade sobre os dados encapsulados. Em nível de abstração mais baixo, associado à implementação, esses procedimentos são implementados por subprogramas denominados *operações*, *métodos* ou *serviços*.

Qualquer processamento a ser realizada sobre os dados encapsulados em um *TAD* só poderá ser executada por intermédio dos procedimentos definidos no modelo matemático do *TAD*, sendo esta restrição a característica operacional mais útil dessa estrutura.

Nesses casos, um programa baseado em *TAD* deverá conter algoritmos e estruturas de dados que implementem, em termos da linguagem de programação adotada, os procedimentos e os modelos de dados dos *TADs* utilizados pelo programa.

Assim, a implementação de cada *TAD* pode ocupar porções bem definidas do programa: uma para a definição das estruturas de dados e outra para a definição do conjunto de algoritmos.

Nessas condições, quaisquer alterações realizadas na estrutura de um dado *TAD* não afetarão as partes do programa que utilizam esse *TAD*.

NÍVEIS DE ABSTRAÇÃO

Uma coleção de atividades, tais como *inserir*, *suprimir* e *consultar*, encapsulada junto com uma estrutura passiva, como um *dicionário* (conjunto de verbetes), pode ser considerada um **tipo abstrato de dados** (*TAD*).

Definido dessa forma, o *TAD DICIONÁRIO* fica representado no nível de abstração mais alto possível: o **nível conceitual**.

Em um nível de abstração mais baixo, denominado **nível de design**, a estrutura passiva deve ser representada por um **modelo de dados** (por exemplo: seqüência ou árvore binária de busca) e as operações devem ser especificadas através de **procedimentos** cuja representação não dependa de uma linguagem de programação.

Em um nível de abstração ainda mais baixo, denominado **nível de implementação**, deve-se tomar como base o design do *TAD* e estabelecer representações concretas para os elementos de sua estrutura em termos de uma linguagem de programação específica.

No exemplo do *TAD DICIONÁRIO*, se o design escolhido para a estrutura passiva for a árvore binária de busca, será possível implementar esse modelo de dados em Scheme, por exemplo, em termos de uma estrutura de dados do tipo *lista* correspondendo à representação prefixada *(raiz sub-árvore-esquerda sub-árvore-direita)*. Os procedimentos serão implementados por meio dos algoritmos apropriados às operações *raiz, esquerda, direita* etc. utilizando-se os recursos disponíveis em Scheme para codificar estruturas ativas (operações e controle). As duas estruturas, a ativa e a passiva, do *TAD* poderão eventualmente constituir um encapsulamento (módulo) específico e identificável no programa construído.

MOTIVAÇÃO

Uma razão importante para programar em termos de *TAD* é o fato de que os elementos da estrutura passiva do *TAD* são acessíveis somente através dos elementos da estrutura ativa.

No caso do *TAD DICIONÁRIO*, por exemplo, o acesso a qualquer elemento da estrutura de dados pode ocorrer apenas via os algoritmos correspondentes às operações *inserir*, *eliminar* e *consultar*.

Essa restrição conduz a uma forma eficiente de **programação defensiva**, protegendo os dados encapsulados no *TAD* contra manipulações inesperadas por parte de outros algoritmos.

Uma segunda razão, também importante, para programar em termos de *TAD* é o fato de que seu uso permite introduzir alterações nas estruturas definidas no nível de implementação — visando, por exemplo, aumento de eficiência — livre da preocupação de gerar erros no restante do programa. Tais erros não ocorrem porque a única conexão entre o *TAD* e o restante do programa é aquela constituída pela interface **imutável** dos algoritmos que implementam a estrutura ativa do *TAD*.

Por fim, pode-se dizer que um *TAD* bem construído pode tornar-se uma porção de código confiável e genérica, permitindo e aconselhando seu **reúso** em outros programas. Dessa forma, aumenta-se a produtividade na construção de programas e, sobretudo, garante-se a qualidade dos produtos gerados.

TIPO ABSTRATO DE DADOS ¾ RESUMO

TAD é uma estrutura de programação que visa implementar:

- o domínio de um modelo matemático de dados
- um conjunto de operações básicas que atuam com exclusividade sobre os valores desse domínio.

Qualquer operação a ser realizada sobre dados definidos por meio dessa estrutura só poderá ser executada por intermédio dos algoritmos definidos no ***TAD***.

Um ***TAD*** deverá ocupar uma porção bem definida do programa, a qual conterá uma estrutura com dois componentes:

- um, associado à definição das estruturas de dados que implementam a representação dos valores que constituem o domínio do modelo matemático de dados considerado
- outro, associado à definição do conjunto de algoritmos que implementam as operações que atuam com exclusividade sobre essas estruturas de dados.

Nessas condições, quaisquer alterações realizadas na implementação da estrutura de um dado ***TAD*** não afetarão as partes do programa que utilizam esse ***TAD***.

TAD É CONCEITO BÁSICO PARA A ABORDAGEM ORIENTADA A OBJETOS

TIPO ABSTRATO DE DADOS
¾ EXEMPLOS DE IMPLEMENTAÇÃO EM SCHEME ¾

RECAPITULANDO

No nível de abstração correspondente à implementação em termos de uma linguagem de programação específica, um **tipo abstrato de dados**, TAD, é uma estrutura de programa contendo:

- uma **representação de dados** de um conjunto de valores (**domínio** do tipo), expressa em termos da linguagem de programação adotada
- o conjunto de algoritmos que atuam **com exclusividade** sobre essa representação de dados e implementam, na linguagem de programação adotada, as **operações básicas** que processam os valores do domínio considerado.

Assim sendo, quaisquer outras áreas de um programa que contenha um TAD não possuirão, e portanto não utilizarão, o conhecimento relativo aos detalhes da representação de dados específica associada àquele domínio de valores, bem como aos detalhes da implementação das operações básicas tornadas disponíveis externamente.

Nessas condições, as áreas do programa usuárias do TAD só poderão processar os valores do domínio associado ao tipo por intermédio das operações básicas contidas no TAD. Portanto, essas áreas permanecerão invariantes em relação a qualquer alteração efetuada seja na representação de dados seja nos algoritmos que implementam as operações básicas do TAD.

TAD CONJUNTO

REPRESENTAÇÃO DO DOMÍNIO

- um conjunto vazio é representado pela lista vazia: { } → ( )
  (define conj-vazio '( ))

- um conjunto não vazio é representado por uma lista, sem itens repetidos, contendo os elementos do conjunto

OPERAÇÕES BÁSICAS

- **teste de conjunto vazio**

  (define conj-vazio? null?)

- **teste de conjunto**

  (define conjunto? list?)

- **cardinalidade de conjunto**

  (define card (lambda (conj) (comprimento-lista conj)))

- **teste de pertinência de item a conjunto**

```
(define pertence-conj? (lambda (item conj)              ; processo iterativo
      (cond ( (conj-vazio? conj) #f)
            ( (equal? (car conj) item) #t)
            (else (pertence-conj? item (cdr conj))) ) ))
```

- **teste de subconjunto** (todo elemento de conj1 pertence a conj2)

```
(define subconjunto? (lambda (conj1 conj2)      ; processo iterativo
    (cond ( (conj-vazio? conj1) #t)
          ( (conj-vazio? conj2) #f)
          ( (not (pertence-conj? (car conj1) conj2)) #f)
          (else (subconjunto? (cdr conj1) conj2)) ) ))
```

♦ **construtor de conjunto unitário**

```
(define conjunto-unitario (lambda (item) (cons item conj-vazio) ))
```

♦ **união de conjuntos** (elementos pertencentes a conj1 ou conj2)

```
(define uniao-rec (lambda (conj1 conj2)              ;  processo recursivo
      (cond ( (conj-vazio? conj1) conj2)
            ( (conj-vazio? conj2) conj1)
            ( (pertence-conj? (car conj1) conj2) (uniao-rec (cdr conj1) conj2))
            (else (cons (car conj1) (uniao-rec (cdr conj1) conj2))) ) ))


(define uniao-it-aux (lambda (conj1 conj2 ac)        ;  processo iterativo
   (if (conj-vazio? conj1)
       ac
       (uniao-it-aux (cdr conj1)
                     conj2
                     (if (pertence-conj? (car conj1) conj2)
                         ac
                         (cons (car conj1) ac) ) ) ) ))

(define uniao-it (lambda (conj1 conj2) (uniao-it-aux conj1 conj2 conj2) ))
```

♦ **interseção de conjuntos** (elementos pertencentes a conj1 e conj2)

```
(define intersecao-rec (lambda (conj1 conj2)          ; processo recursivo
    (cond ( (conj-vazio? conj1) conj-vazio)
          ( (conj-vazio? conj2) conj-vazio)
          ( (pertence-conj? (car conj1) conj2)
            (cons (car conj1) (intersecao-rec (cdr conj1) conj2)) )
          (else (intersecao-rec (cdr conj1) conj2)) ) ))

(define intersecao-it-aux (lambda (conj1 conj2 ac)  ; processo iterativo
    (if (conj-vazio? conj1)
        ac
        (intersecao-it-aux (cdr conj1)
                           conj2
                           (if (pertence-conj? (car conj1) conj2)
                               (cons (car conj1) ac)
                               ac ) ) ) ))

(define intersecao-it (lambda (conj1 conj2)
    (intersecao-it-aux conj1 conj2 conj-vazio) ))
```

♦ **diferença de conjuntos** (elementos de conj1 não contidos em conj2)

```
(define diferenca-rec (lambda (conj1 conj2)          ; processo recursivo
      (cond ( (conj-vazio? conj1) conj-vazio)
            ( (conj-vazio? conj2) conj1)
            ( (not (pertence-conj? (car conj1) conj2))
              (cons (car conj1) (diferenca-rec (cdr conj1) conj2)) )
            (else (diferenca-rec (cdr conj1) conj2)) ) ))


(define diferenca-it-aux (lambda (conj1 conj2 ac)    ; processo iterativo
     (if (conj-vazio? conj1)
         ac
         (diferenca-it-aux (cdr conj1)
                           conj2
                           (if (pertence-conj? (car conj1) conj2)
                               ac
                               (cons (car conj1) ac) ) ) ) ))

(define diferenca-it (lambda (conj1 conj2)
       (diferenca-it-aux conj1 conj2 conj-vazio) ))
```

USO DO TAD CONJUNTO

♦ **função para acrescentar um dado item a um conjunto dado**

```
(define acrescenta (lambda (item conj)
      (uniao (conjunto-unitario item) conj) ))
```

♦ **predicado para verificar de dois conjuntos dados são iguais**

```
(define conj-iguais? (lambda (conj1 conj2)
      (and (subconjunto? conj1 conj2 )
           (subconjunto? conj2 conj1 ) ) ))
```

♦ **função para retirar os elementos do conjunto *a* contidos no conjunto *b* e dobrar de valor os elementos restantes**

Por exemplo: (limpa-e-dobra '(1 2 3) '(4 1 3 5)) ⇒ (8 10)

```
(define limpa-e-dobra (lambda (a b)
    (mapeia (lambda (n) (* 2 n))
            (diferenca b a) ) ))
```

TAD ÁRVORE BINÁRIA

REPRESENTAÇÃO DO DOMÍNIO

O conceito matemático de **conjunto** está na base da definição de árvore binária.

Em Scheme, a representação de dados empregada para conjuntos é a **lista**, tipo de dados predefinido nessa linguagem.

Assim sendo, e tendo em vista a estrutura de uma árvore binária — constituída por um nodo **raiz**, uma **sub-árvore à esquerda** e uma **sub-árvore à direita** —, há três alternativas para a representação de dados do domínio do tipo *árvore binária*:

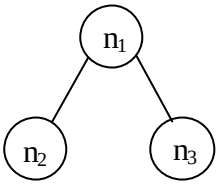
- representação prefixada, por meio da lista *(raiz sub-árvore-esquerda sub-árvore-direita)*
- representação infixada, por meio da lista *(sub-árvore-esquerda raiz sub-árvore-direita)*
- representação pós-fixada, por meio da lista *(sub-árvore-esquerda sub-árvore-direita raiz)*.

Admite-se a existência de uma árvore vazia e, visando proporcionar maior legibilidade aos programas, define-se a constante *arv-bin-vazia*: (define arv-bin-vazia '( )).

Visando simplificar a representação de árvores binárias, adota-se a convenção de representar uma árvore cujas sub-árvores à esquerda e à direita são ambas vazias apenas pelo elemento associado a sua raiz. Ou seja, uma folha é representada pelo elemento associado ao nodo correspondente:

- a árvore binária 

é representada por *n*, e não pela lista (*n* ( ) ( )) considerando-se a forma prefixada.

- a árvore binária 

é representada, utilizando-se a forma prefixada, pela lista (*n1 n2 n3*), e não pela lista (*n1* (*n2* ( ) ( )) (*n3* ( ) ( ))).

OPERAÇÕES BÁSICAS (representação prefixada)

- **testar árvore binária vazia** — predicado *arv-bin-vazia?* —
  (define arv-bin-vazia? null?)

♦ **testar folha** — predicado *folha?* —

```
(define  folha?  (lambda  (ab)  (not  (list?  ab)) ))
```

♦ **criar árvore binária** — função *cria-arvore* —

```
(define  cria-arvore  (lambda  (r  e  d)
      (if  (and  (arv-bin-vazia?  e)  (arv-bin-vazia?  d))
           r
           (cons  r  (cons  e  (cons  d  '( )))) ) ))
```

exemplos

```
(cria-arvore  'r  '( )  '( ))  ⇒  r
(cria-arvore  'r  'e  'd)  ⇒  (r  e  d)
(cria-arvore  'r  '(e  e-fe  e-fd)  'd)  ⇒  (r  (e  e-fe  e-fd)  d)
```

♦ **acessar a raiz** — função *raiz* —

```
(define  raiz  (lambda  (ab)
      (if  (folha?  ab)
           ab
           (car  ab) ) ))
```

♦ **acessar a sub-árvore à esquerda** — função *esquerda* —

```
(define  esquerda  (lambda  (ab)
      (if  (folha? ab)
           arv-bin-vazia
           (cadr  ab) ) ))
```

♦ **acessar a sub-árvore à direita** — função *direita* —

```
(define  esquerda  (lambda  (ab)
      (if  (folha? ab)
           arv-bin-vazia
           (caddr  ab) ) ))
```

Assim sendo, o **conhecimento externo** referente ao TAD *Árvore Binária* refere-se:

- à **representação do domínio,** na forma **prefixada,** limitada à
  - constante *arvore-vazia* que designa uma árvore binária vazia
  - estrutura recursiva de lista ternária *(raiz sub-árvore-esquerda sub-árvore-direita)* usada para representar uma árvore binária não vazia
  - convenção de representar um nodo-folha pelo elemento a ele associado

- às **operações básicas** que atuam com exclusividade sobre essa representação do domínio, implementadas pelas funções definidas precedentemente visando:
  - **testar árvore vazia**
  - **criar árvore**
  - **acessar sub-árvore à esquerda**
  - **testar folha**
  - **acessar raiz**
  - **acessar sub-árvore à direita.**

TAD ÁRVORE BINÁRIA
(representação infixada)

REPRESENTAÇÃO DO DOMÍNIO

- constante *arv-bin-vazia* que designa uma árvore binária vazia
  (define arv-bin-vazia '( ))
- estrutura recursiva de lista ternária
  (sub-árvore-esquerda raiz sub-árvore-direita)
  para representar uma árvore binária não vazia
- convenção de representar um nodo-folha pelo elemento a ele associado

OPERAÇÕES BÁSICAS

- **testar árvore binária vazia**

```
(define arv-bin-vazia? null?)
```

- **testar folha**

```
(define folha? (lambda (ab)
        (not (list? ab)) ))
```

- **criar árvore binária**

```
(define cria-arvore-in (lambda (e r d)
       (if (and (arv-bin-vazia? e) (arv-bin-vazia? d))
           r
           (cons e (cons r (cons d '( )))) ) ))
```

exemplos

(cria-arvore-in '( ) 'r '( )) ⇒ r

(cria-arvore-in 'e 'r 'd) ⇒ (e r d)

(cria-arvore-in '(e-fe e e-fd) 'r 'd) ⇒ ((e-fe e e-fd) r d)

♦ **acessar raiz**

```
(define   raiz-in   (lambda  (ab)
        (if   (folha?  ab)
              ab
              (cadr  ab) ) ))
```

♦ **acessar sub-árvore à esquerda**

```
(define   esquerda-in   (lambda  (ab)
       (if   (folha?  ab)
             arv-bin-vazia
             (car  ab) ) ))
```

♦ **acessar sub-árvore à direita**

```
(define   direita   (lambda  (ab)
      (if   (folha?  ab)
            arv-bin-vazia
            (caddr  ab) ) ))
```

USO DO TAD ÁRVORE BINÁRIA

- **função que receba uma árvore binária e verifique se um dado item está associado a um de seus nodos**

```
(define  pertence-AB?  (lambda (item  ab)
   (cond  ( (arv-bin-vazia?  ab)  #f)
          ( (equal?  (raiz  ab)  item)  #t)
          (else  (or  (pertence-AB?  item  (esquerda  ab))
                 (pertence-AB?  item  (direita  ab)) )) ) ))
```

- **função que receba uma árvore binária não vazia e retorne quantos nodos são folhas**

versão 1

```
(define  conta-folhas-aux  (lambda (ab)
     (cond  ( (arv-bin-vazia?  ab)  0)
            ( (folha?  ab)  1)
            (else  (+  (conta-folhas-aux  (esquerda  ab))
                       (conta-folhas-aux  (direita ab)) )) ) ))

(define  conta-folhas  (lambda (ab)
     (cond  ( (arv-bin-vazia? ab) 0)
        (if  (arv-bin-vazia?  ab)
             "entrada inválida:  árvore vazia"
             (conta-folhas-aux  ab) ) ))
```

versão 2

```
(define  conta-folhas  (lambda (ab)
   (cond  ( (folha?  ab)  1)
          ( (arv-bin-vazia?  (esquerda  ab))  (conta-folhas  (direita  ab)) )
          ( (arv-bin-vazia?  (direita  ab))  (conta-folhas  (esquerda  ab)) )
          (else  (+  (conta-folhas  (esquerda  ab))
                     (conta-folhas2  (direita  ab)) )) ) ))
```

TAD MATRIZ

REPRESENTAÇÃO DE DADO

- arranjo bidimensional cujos elementos são arranjos unidimensionais, cada um representando uma linha da matriz
- matriz (m x n) , contendo m linhas e n colunas

#(#(a_{00} a_{01} ... $a_{0(n-1)}$)
#(a_{10} a_{11} ... $a_{1(n-1)}$)
.....
#($a_{(m-1)0}$ $a_{(m-1)1}$ ... $a_{(m-1)(n-1)}$))

OPERAÇÕES BÁSICAS

- **testar matriz vazia**

```
(define vetor-vazio? (lambda (vet) (zero? (vector-length vet)) ))

(define matriz-vazia? (lambda (mat) (vetor-vazio? mat) ))
```

- **retornar o número de linhas de uma matriz**

```
(define num-linhas-mat (lambda (mat) (vector-length mat) ))
```

- **retornar o número de colunas de uma matriz**

```
(define num-colunas-mat (lambda (mat) (vector-length (vector-ref mat 0)) ))
```

♦ **criar matriz de zeros de dimensão m x n**

```
(define  cria-vetor-aux  (lambda  (vet  f  ind)
    (if  (>=  ind  0)
        (begin  (vector-set!  vet  ind  (f  ind))
                (cria-vetor-aux  vet  f  (-  ind  1)) )
        vet ) ) )

(define  cria-vetor  (lambda  (z  f)      ;  cria vetor de dimensão z onde cada elemento
                                          ;  é uma dada função f de seu índice
     (cria-vetor-aux  (make-vector  z)  f  (-  z  1)) ))


(define  make-matrix  (lambda  (m  n)   ;  cria matriz de zeros de dimensão m x n
    (cria-vetor  m
                 (lambda  (x)  (cria-vetor  n
                                            (lambda  (y)  0) )) ) ))
```

♦ **acessar o elemento ij de uma matriz**

```
(define  matrix-ref  (lambda (mat  i  j)  (vector-ref  (vector-ref  mat  i)  j) ))
```

♦ **alterar para um valor dado o elemento ij de uma matriz**

```
(define  matrix-set!  (lambda  (mat  i  j  val)  (vector-set!  (vector-ref  mat  i)  j  val) ))
```

USO DO TAD MATRIZ

Função para criar e retornar uma matriz de dimensão m x n onde cada elemento seja igual a uma função de seus índices.

```
(define  cria-matriz-aux  (lambda (mat f m n i j)
    (cond  ( (= i m) mat)
           ( (< j n)
            (begin  (matrix-set! mat i j (f i j))
                    (cria-matriz-aux mat f m n i (+ j 1)) ) )
           (else (cria-matriz-aux mat f m n (+ i 1) 0)) ) ))

(define  cria-matriz  (lambda (m n f)
     (cria-matriz-aux (make-matrix m n) f m n 0 0) ))
```

Função *cria-exibe-mat+diag* **para criar uma matriz M(nxn) onde cada elemento seja igual a uma função específica de seus índices, para exibi-la de modo que cada linha ocupe uma linha da tela e para exibir uma seqüência contendo os elementos da diagonal de M.**

Exemplificar usando o formato de apresentação indicado a seguir, resultado de uma chamada (cria-exibe-mat+diag 3 + ‘soma).

```
MATRIZ M (3 3 soma)
0 1 2
1 2 3
2 3 4
ELEMENTOS DA DIAGONAL
0 2 4
```

```
(define exibe-mat (lambda (mat n i j)
    (cond ( (= i n) )
          ( (< j n)
            (begin (display (matrix-ref mat i j))
                   (display " ")
                   (if (= j (- n 1)) (newline))
                   (exibe-mat mat n i (+ j 1)) ) )
          (else (exibe-mat mat n (+ i 1) 0)) ) ))

(define exibe-diag (lambda (mat n i j)
    (cond ( (= i n) )
          ( (< j n)
            (begin (display (matrix-ref mat i j))
                   (display " ")
                   (exibe-diag mat n (+ i 1) (+ j 1)) ) )
          (else (exibe-diag mat (+ i 1) 0)) ) ))

(define cria-exibe-mat+diag (lambda (n f op)
   (let ( (M (cria-matriz n n f)) )
        (begin (display "MATRIZ M (") (display n) (display " ") (display n)
               (display " ") (display op) (display ")")
               (newline)
               (exibe-mat M n 0 0)
               (display "ELEMENTOS DA DIAGONAL")
               (newline)
               (exibe-diag M n 0 0)
               (newline) ) ) ))
```