Vetores e Matrizes

Linguagem Haskell

Maria Adriana Vidigal de Lima

Faculdade de Computação - UFU

Setembro - 2009

Fundamentos

- Um vetor é um conjunto finito de elementos homogêneos. Os elementos são ordenados de forma que exista um primeiro elemento, um segundo, e assim por diante.
- Para alcançar qualquer elemento, utilizamos sua posição no vetor.
- A posição inicial do vetor e a posição final são definidas no programa e não são alteradas durante a execução.

Lista x Vetor

Uma lista é composta sempre de dois segmentos: cabeça (*head*) e corpo (*tail*). A cabeça da lista é sempre o primeiro elemento. Um determinado elemento pode ser obtido por sua posição.

```
> 'a':['b','c','d']
"abcd"
> ['a','b','c','d']!!2
'c'
> [32,6,1,8,43]!!0
32
```

Vetores

A linguagem Haskell possui um tipo de dados e algumas funções pré-definidas para a manipulação de vetores, contidos no módulo **Array**. Portanto, é necessário importar tal módulo:

```
import Array
```

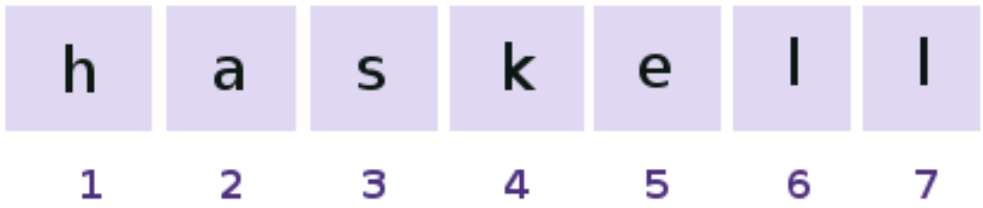
Para a criação de um vetor, usamos a função `array` (contida no módulo **Array**) da seguinte forma:

- O primeiro argumento define a tupla `(linhas,colunas)`.
- O segundo argumento é a lista de associações da forma `(indice,valor)`.

Listas em Haskell

A declaração de tipos para o vetor `vet` indica que os índices do vetor são valores inteiros e os seus elementos são caracteres.

```
vet::Array Int Char
vet = array (1,7) [(1,'h'),(2,'a'),(3,'s'),(4,'k'),
                   (5,'e'),(6,'l'),(7,'l')]
```



Indices no Vetor

Um índice não pode aparecer mais de uma vez no vetor, ou estar fora do limite definido.

```
vet::Array Int Char
vet = array (1,7) [(2,'a'),(5,'e'),(3,'s'),(1,'h'),(4,'k'),
                   (6,'l'),(7,'l')]

vet1::Array Int Int
vet1 = array (-1,3) [(-1,2),(0,4),(1,10),(2,13),(3,39)]
```

Construção de Vetor

Um vetor pode ser preenchido de modo indireto, com o auxílio de funções construtoras, tais como aquelas definidas através de listas por compreensão:

```
quadrados :: Int -> Array Int Int
quadrados n = array (0,n) [(i,i*i)| i <- [0..n]]
```

```
Main> quadrados 5
array (0,5) [(0,0),(1,1),(2,4),(3,9),(4,16),(5,25)]
```

Operador de acesso

Pode-se ter acesso à um elemento no vetor conhecendo-se sua posição. Assim, através do uso do operador ! temos:

```
vet::Array Int Char
vet = array (1,7) [(1,'h'),(2,'a'),(3,'s'),(4,'k'),(5,'e'),
                   (6,'l'),(7,'l')]

> vet!5
'e'
> vet!0
Program error: Ix.index: index out of range
```

Função `bounds`

A função `bounds` permite que obtenhamos a tupla que define o tamanho do vetor: (inicio, fim).

```
vet1::Array Int Int
vet1 = array (-1,3) [(-1,2),(0,4),(1,10),(2,13),(3,39)]

> bounds vet1
(-1,3)

> bounds (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
(0,3)
```

Função `elems`

A função `elems` permite que os elementos do vetor sejam obtidos (em ordem) e mantidos numa lista.

```
> elems (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
[5,2,3,1]

> elems vet
"haskell"
```

Resumo

Funções pré-definidas para vetores:

```
> array (0,3) [(2,3),(1,2),(0,5),(3,1)]
array (0,3) [(2,3),(1,2),(0,5),(3,1)]

> (array (0,3) [(2,3),(1,2),(0,5),(3,1)]) ! 2
3

> bounds (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
(0,3)

> elems (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
[5,2,3,1]
```

Índices de um vetor

Para criar a lista dos índices de um vetor, podemos usar a função `range`:

```
> bounds (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
(0,3)

> range (0,3)
[0,1,2,3]

> range (-3,12)
[-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12]

> range (bounds (array (0,3) [(2,3),(1,2),(0,5),(3,1)]))
[0,1,2,3]
```

Obtendo os elementos de um vetor

Para obtemos os elementos de um vetor podemos escrever a seguinte função:

```
lista:: Array Int a -> [a]
lista b =  [b!a| a <- range (bounds b)]
```

```
> bounds (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
(0,3)
> range (0,3)
[0,1,2,3]
> lista (array (0,3) [(2,3),(1,2),(0,5),(3,1)])
[5,2,3,1]
> lista (array (0,3) [(2,'a'),(1,'b'),(0,'c'),(3,'d')])
"cbad"
> elems (array (0,3) [(2,'a'),(1,'b'),(0,'c'),(3,'d')])
"cbad"
```

Função genérica para a criação de vetor

A função `criaVet` possibilita a criação de vetor a partir da aplicação de cada índice a uma função `f`:

```
criaVet :: (Ix a) => (a -> b) -> (a,a) -> Array a b
criaVet f interv =  array interv [(i, f i) | i <- range interv]
```

```
> criaVet (+1) (1,4)
array (1,4) [(1,2),(2,3),(3,4),(4,5)]

> criaVet (^2) (1,4)
array (1,4) [(1,1),(2,4),(3,9),(4,16)]

> criaVet (chr) (65,70)
array (65,70) [(65,'A'),(66,'B'),(67,'C'),(68,'D'),
               (69,'E'),(70,'F')]
```

Sequência de Fibonacci

Os vetores podem ser definidos recursivamente, isto é, o valor de um elemento depende do valor de outro elemento. A função que retorna um vetor com a sequência de Fibonacci é um exemplo:

```
fibs    :: Int -> Array Int Int
fibs n = a
         where a = array (0,n) ([(0, 1), (1, 1)] ++
                                [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

```
> fibs 5
array (0,5) [(0,1),(1,1),(2,2),(3,3),(4,5),(5,8)]

Main> fibs 10
array (0,10) [(0,1),(1,1),(2,2),(3,3),(4,5),(5,8),
              (6,13),(7,21),(8,34),(9,55),(10,89)]
```

Construção de Matrizes

Uma matriz é representada por um vetor de duas dimensões, e é construída pela função `array` com uma *tupla-2*:

```
mat = array ((1,1),(2,3)) [((1,1),4), ((1,2),0), ((1,3),3),
                           ((2,1),5), ((2,2),1), ((2,3),4)]
```

Em ((1,1),(2,3)), a primeira tupla (1,1) define o primeiro índice de linha, e o primeiro índice de coluna, e a segunda tupla (2,3) os limites da matriz.

Operações sobre Matrizes

Podemos aplicar sobre matrizes as mesmas operações que utilizamos para vetores:

```
mat = array ((1,1),(2,3)) [((1,1),4), ((1,2),0), ((1,3),3),
                           ((2,1),5), ((2,2),1), ((2,3),4)]

> mat!(2,2)
1

> bounds mat
((1,1),(2,3))

> elems mat
[4,0,3,5,1,4]
```

Função Soma para Matrizes

A função somaMat produz uma nova matriz em que cada posição é a soma dos elementos nas posições correspondentes de duas matrizes de entrada (do mesmo tamanho):

```
mat = array ((1,1),(2,3)) [((1,1),4), ((1,2),0), ((1,3),3),
                           ((2,1),5), ((2,2),1), ((2,3),4)]

somaMat::(Ix a,Num b) => Array (a,a) b -> Array (a,a) b ->
                         Array (a,a) b
somaMat mat1 mat2 = array (bounds mat1)
                    [((i,j), mat1!(i,j) + mat2!(i,j)) |
                      (i,j) <- range (bounds mat1)]

> somaMat mat mat
array ((1,1),(2,3)) [((1,1),8),((1,2),0),((1,3),6),
                     ((2,1),10),((2,2),2),((2,3),8)]
```

Bibliografia

1. *Haskell - Uma abordagem prática*. Cláudio César de Sá e Márcio Ferreira da Silva. Novatec, 2006.
2. *A Gentle Introduction to Haskell. http://www.haskell.org/tutorial/arrays.html*